
LA QUALITÉ EN C++
PHILIPPE PRADOS
. RÈGLES DE CODAGE
. PATTERNS DE PROGRAMMATION
. FONCTIONNEMENT DU COMPILATEUR
EYROLLES

TABLE DES MATIERES

AVANT - PROPOS

La « qualité logiciel » est un élément crucial pour les entreprises. Le coût d'une application est fortement dépendant de sa qualité. Le C++ est un langage très intéressant pour le développeur. La qualité est difficilement maîtrisable avec ce type de langage, car beaucoup de traitements sont gérés par le compilateur « derrière le dos » du développeur.

Ce livre permettra aux développeurs de programmer avec rigueur et aux entreprises de rédiger la « charte de développement » pour les applications C++. Il s'adresse aussi aux développeurs désirant en savoir plus sur ce langage. Il indique les règles à suivre pour développer sans erreurs, ou pour localiser celles-ci plus rapidement. Il introduit des *patterns* pour traduire certains concepts absents dans le langage. De plus, il explique quelle technique est utilisée par le compilateur pour gérer l'héritage ou le polymorphisme et « comment ça marche ».

Il faut connaître le C++ pour aborder l'ouvrage. Vous trouverez facilement de très bons ouvrages pour vous initier à ce langage. Je ne peux que vous conseiller l'ARM (*Annoted C++ Reference Manual* [Stroustrup, ARM:94]), ce document servant de référence au comité de normalisation du langage. « La qualité en C++ » vous permettra, par la suite, de vous perfectionner pour devenir un expert. Ainsi, vous parlerez le C++ *sans accent* !

Le C++ est une évolution du langage C amenant pour le programmeur une puissance nouvelle. L'approche objet permet de faire abstraction de beaucoup de détails répétitifs pour ne se soucier que de l'essentiel. Lors de l'écriture d'une simple expression comme

`a=b*c+d`, une succession d'étapes est nécessaire à la résolution de celle-ci. Beaucoup de concepts sont mis en oeuvre. Ceux-ci sont devenus tellement naturels qu'il est facile d'en faire abstraction et de laisser le compilateur s'en occuper. Il n'est pas nécessaire de les maîtriser précisément, car en général, tous fonctionnent comme on l'imagine. Malheureusement, plus le langage permet de s'affranchir de détails, plus ceux-ci peuvent avoir des effets pervers.

Le langage C est très proche de l'assembleur, c'est pour cela qu'il a été choisi par l'ensemble de l'industrie. Il est possible d'écrire un système d'exploitation avec le C (exemple Unix). Il existe peu de langages évolués ayant permis d'écrire un Système d'exploitation. Il n'est pas nécessaire de connaître l'assembleur pour maîtriser le C, pourtant, certains effets de bord ne s'expliquent que parce qu'il y a l'assembleur *derrière* le C. Il est courant de penser à l'assembleur généré lorsque l'on développe en C ; pas un assembleur particulier, mais les concepts présents dans chaque assembleur. Les notions de mémoire globale, de pile et de pointeurs doivent être maîtrisées pour écrire en C.

Pour le C++, l'abstraction est énorme. Lorsqu'on écrit en C++, on devrait toujours penser au C généré. C'est un peu le paradoxe de ce type de langage, plus celui-ci permet de s'affranchir de détails, plus il faut être capable de les maîtriser. Ce livre est fait pour vous informer sur les subtilités du C++.

Comme beaucoup de langages, l'expérience est transmise par le « bouche à oreille ». C'est cette connaissance, fruit de l'expérience, que je me propose de vous transmettre.

Mon objectif est de vous expliquer *comment* utiliser le langage en respectant des règles de qualité. La plupart des ouvrages vous expliquent chaque concept les uns après les autres, mais n'informent pas sur les pièges des interactions entre ces concepts.

Le C++ offre beaucoup d'outils permettant de faciliter les développements. Ceux-ci doivent être correctement utilisés pour ne pas laisser des erreurs à retardement dans vos programmes. Ce n'est pas parce que votre programme *semble* fonctionner, qu'il n'est pas criblé d'erreurs. Celles-ci apparaîtront lors des situations les plus gênantes. Selon la « loi de Murphy », les erreurs n'apparaissent jamais au développeur, ce sont les utilisateurs qui les rencontrent. Chacun ayant vécu cette expérience, sait qu'une démonstration ne fonctionne jamais, c'est le fameux « effet démo ». Pour vous éviter ce type de problème, je vous conseille de suivre les recommandations indiquées dans ce livre.

Négliger les points évoqués amène un comportement imprévisible de vos programmes. Si vous utilisez le C++ à tort et à travers, votre code sera inefficace, impossible à modifier et à maintenir. Il faut rédiger avec beaucoup de rigueur vos programmes pour augmenter leur robustesse et leur qualité.

La structure de ce livre reprend les différentes étapes d'un développement :

Le chapitre deux, « Conception d'implémentation », précise la bonne utilisation du langage. Quelles questions faut-il se poser pour écrire une classe et une méthode ? Après les phases

de conception et de conception détaillée, il faut passer au codage. C'est une dernière chance de pouvoir vérifier la conception.

Le chapitre trois, « *Patterns* de programmation », indique comment traduire correctement certains concepts. Le C++ ne possède pas une traduction de tous les concepts utiles au développement. Les « *Patterns* de programmation » proposent une traduction validée pour combler ces manques. La conception détaillée doit tenir compte de ces traductions génériques.

Le chapitre quatre, « Autres principes de codage », renseigne sur des principes de codage qui ne sont pas des traductions de concept.

Le chapitre cinq, « Outils de debug », propose des outils pour faciliter la détection d'erreurs.

La chapitre six indique l'ensemble des « Règles » à appliquer pour bien développer en C++. Ce chapitre doit servir de référence. Les explications détaillées de certaines d'entre elles, sont reportées au chapitre sept : « Description des règles ». J'ai choisi cette approche pour ne pas alourdir le chapitre de référence. Lorsque vous aurez compris les justifications des règles, vous n'aurez plus à consulter celui-ci.

Le chapitre huit, « Tests », donne la manière de vérifier une classe et ses méthodes. Ce chapitre traite des préconditions, des postconditions et des invariants. Il propose une démarche pour effectuer les tests unitaires d'une classe et les tests d'un paquet de classes.

Le chapitre neuf et dernier, intitulé « Comment ça marche » explique les techniques employées par le compilateur pour traduire votre programme, il permet de mieux comprendre les effets de bord possibles de celui-ci.

Un index, à la fin du document, permet de remonter aux différents chapitres. Si vous constatez une erreur dans votre programme, consultez l'index pour voir si celle-ci ne correspond pas à un cas déjà référencé.

La notation choisie pour représenter les relations entre les classes est celle proposée par Rumbaugh, dans « *OMT Modélisation et conception orientées objet* » [Rumbaugh et al, OMT:95]. Vous trouverez en Annexe, un résumé de cette notation.

Je voudrais remercier les personnes qui ont rendu possible la réalisation de ce livre, particulièrement mon épouse qui a supporté les nombreuses relectures du manuscrit. Tous mes remerciements à Joël Deslandes et Laurent Kurylo d'Arthur Andersen pour leurs révisions et la pertinence de leurs commentaires ainsi que tous ceux qui m'ont encouragé pour la réalisation de ce livre.

Vous pouvez me faire part de vos suggestions à l'adresse Internet suivante : `pprados@club-internet.fr`

CHAPITRE 1

RAPPELS DE CONCEPTS DE BASE

Pour pouvoir comprendre les informations présentes dans ce livre, il faut connaître bien entendu le C++. Nous vous présentons néanmoins dans ce chapitre, un rappel des concepts de base manipulés tout au long de cet ouvrage.

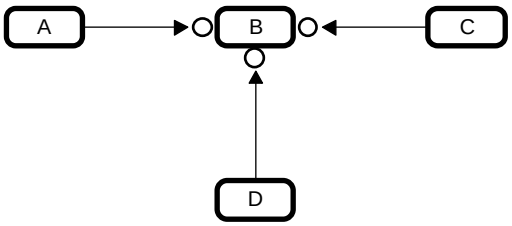
Les schémas permettent de résumer les concepts décrits. Seul les concepts étant utiles à la compréhension de l'ouvrage sont repris :

relation, agrégation, fabricant, conversion par construction.

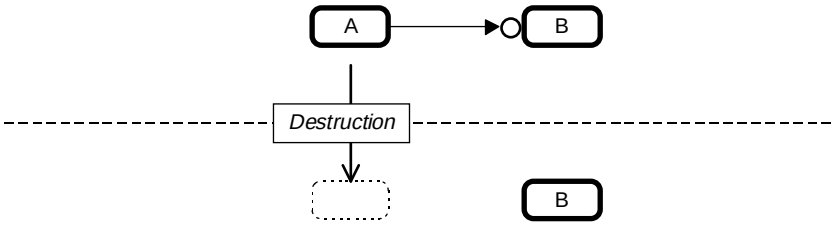
A. RELATION

Une relation entre deux classes indique une dépendance entre deux ou plusieurs instances. En C++, cela se traduit par un pointeur.

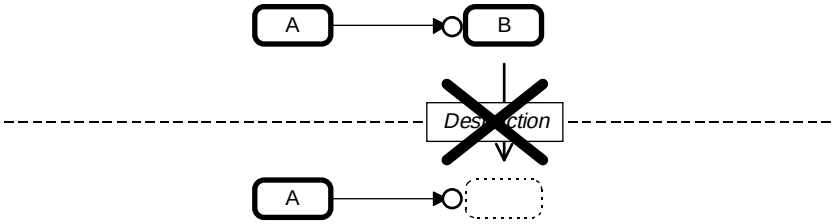
- Un objet pointé peut être partagé par plusieurs objets.



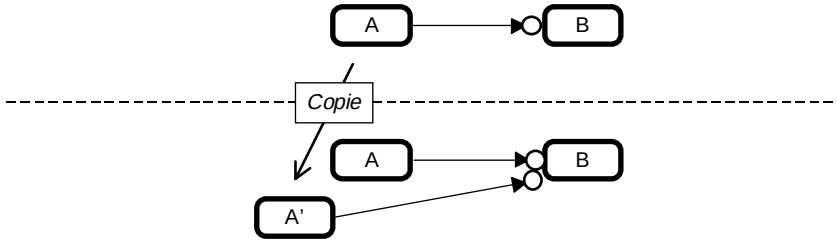
- La destruction de A n'entraîne pas la destruction de B.



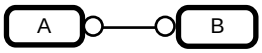
- Il est interdit de détruire l'objet B s'il existe des objets en relation avec lui. Sinon, l'objet A aurait un pointeur dans le vide.



- La copie de A n'entraîne pas la copie de B.



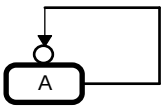
- Une relation est bidirectionnelle si les objets sont en relation entre eux.



En C++, cela se traduit par un pointeur dans chaque classe.

```
class B;
class A
{ B* _ptb;
};
class B
{ A* _pta;
};
```

Une classe peut avoir un pointeur sur un objet du même type.




```
class A
{ A* _pta;
};
```

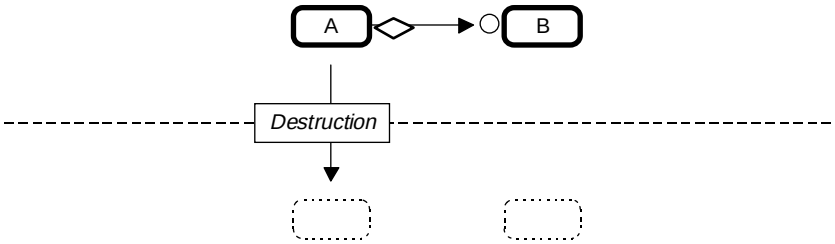
La relation est également appelée « relation *use* ».

B. AGREGATION

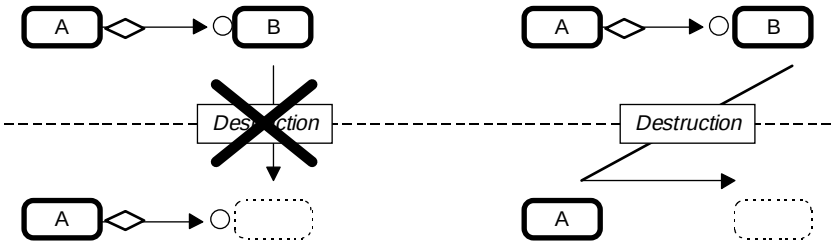
Une agrégation est une relation particulière. Un objet en agrégation n'existe que par la présence de l'objet propriétaire. L'agrégation ajoute sur la relation, une information de durée de vie. Tous les attributs d'une classe sont des agrégations car ils sont détruits en même temps que l'instance les contenant. Si un objet en relation est détruit lors de la destruction de l'objet contenant, il s'agit d'une agrégation. Le document de normalisation ANSI/ISO du C++ parle de « *subobjets* ».

- La durée de vie de B dépend de la durée de vie de A ou de C.

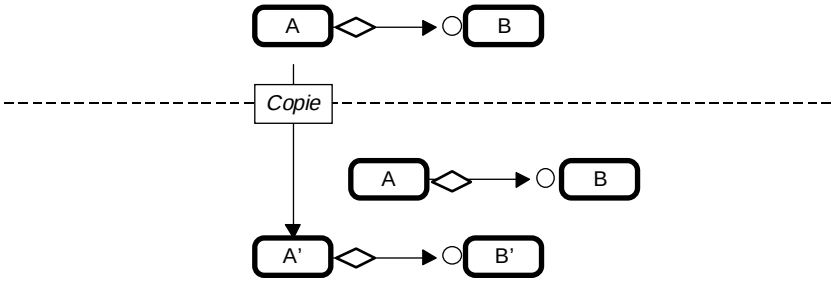
- La destruction de A entraîne la destruction de B



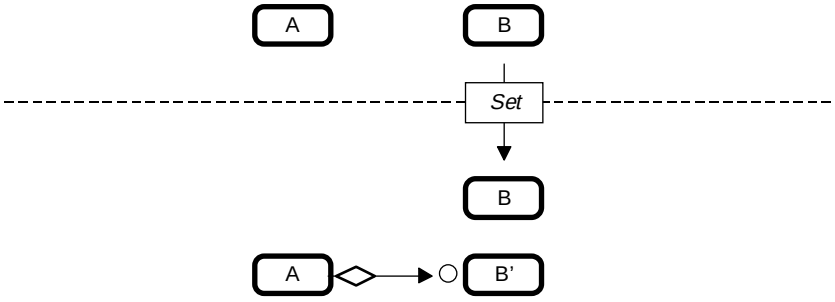
- Il n'est pas possible de détruire directement une agrégation. Il faut le demander à l'agrégeant.



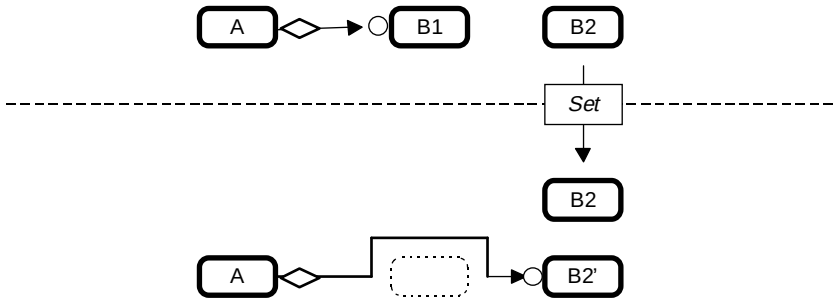
- La copie de A entraîne la copie de B



- Lors d'une valorisation d'un attribut, A garde une copie de l'objet reçu en paramètre.

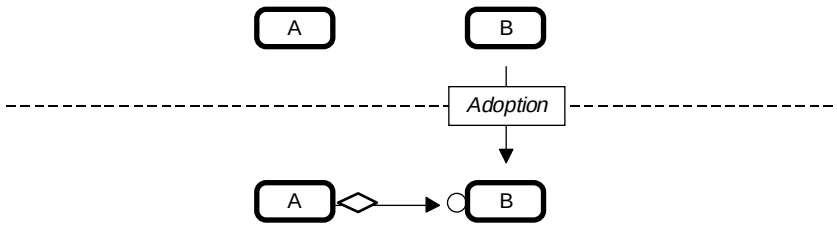


- La valorisation d'un objet agrégé détruit l'agrégation précédente. Cela est équivalent à l'affectation d'un attribut.

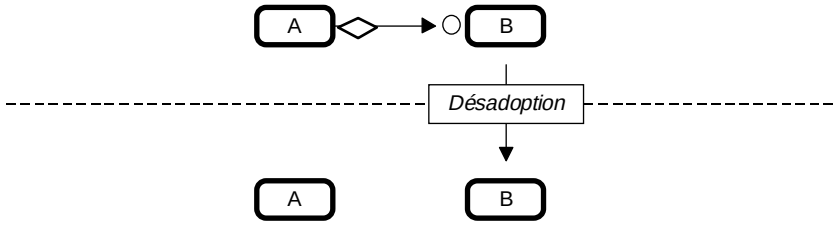


L'adoption d'un pointeur indique qu'un objet prend en charge celui-ci et se chargera de sa destruction. Un pointeur peut transiter de propriétaire en propriétaire par adoptions successives. Bien maîtriser l'adoption des pointeurs permet de gérer correctement les allocations mémoires.

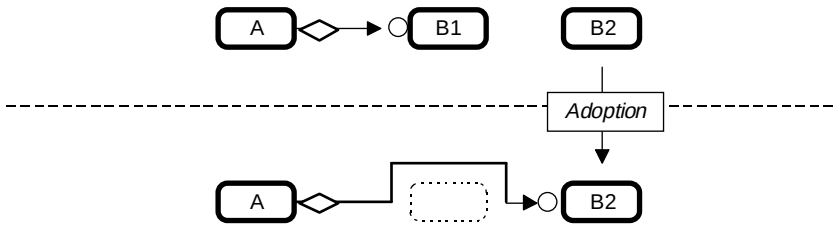
- L'adoption est un transfert de responsabilité sur la durée de vie d'un objet. Il n'est possible d'adopter un objet que si il est libre (sans propriétaire).



- La désadoption est un abandon de responsabilité



- Si la cardinalité de l'agrégation est de un, l'adoption détruit l'agrégation précédente.



La rédaction en C++ d'une agrégation peut se traduire de deux façons différentes. Soit en déclarant un objet dans un objet,

```
class A
{ //...
};
class B
{ A _agregation;
};
```

soit en déclarant un pointeur, et en ajoutant dans le destructeur de la classe agrégeante, la destruction de l'objet agrégé (Voir aussi « Durée de vie des objets », page 57).

```
class A
{ //...
};
class B
{ A* _agregation;
  public:
    B(A* agregation)                        // Adoption du pointeur
  : _agregation(agregation)
  { }
    B(const A& agregation)                  // Copie de l'objet
  : _agregation(new A(agregation))
  { }
    ~B()
  { delete _agregation; }
};
```

L'implantation par un pointeur permet de bénéficier du polymorphisme et du concept d'adoption. Que l'agrégation soit implantée *via* un attribut ou un pointeur, l'interface doit être identique.

```
class A
{ B _b;
  public:
  void setb(const B& b)
  { _b=b; }
};
```

ou

```
class A
{ B* _b;
  public:
    A() : _b(new B()) { } // Initialiser l'agrégation
    ~A() { delete _b; }   // Détruire l'agrégation
  void setb(const B& b)   // Positionner l'agrégation
  { delete _b;            // Détruire l'ancien, puis
    _b=new B(b);          // faire une copie du nouveau B
  }
};
```

Le choix d'implantation ne doit pas impacter l'interface de l'objet. L'agrégation est également appelée « relation *has* ».

C. FABRICANT

Une méthode ou fonction « fabricante » (*factory* en anglais) est une méthode construisant un objet dans le tas et renvoyant l'adresse de cet objet. L'appelant de la méthode ou de la fonction devient alors propriétaire de l'objet. Il devra s'occuper de le détruire.

```
A* MakeObjA()
{ return new A(); }

void main()
{ A* pA=MakeObjA();
  delete pA;
}
```

Un pointeur retourné par une méthode « fabricante » doit toujours être *adopté*.

D. CONVERSION PAR CONSTRUCTION

Pour convertir un objet, il existe deux techniques. La conversion classique utilise un opérateur de conversion.

```
class A
{ //...
};
class B
{ public:
    operator A() const
  { // Operateur de conversion
    return A(...
  }
};
```

L'opérateur de conversion de la classe `B` convertit une instance `B` en instance `A`. Une autre technique pour convertir un objet est d'utiliser un constructeur n'ayant qu'un seul paramètre.

```
class B
{ //...
};
class A
{ A();
  A(const B& x) // Conversion par construction
  { //...
  }
};
```

Le compilateur peut alors construire un objet temporaire pour convertir un objet.

```
void f(const A& a);

void main()
{ B b;
  f(b); // Conversion par construction
}
```

L'appel de `f(b)` est converti par le compilateur comme cela :

```
void main()
{ B b;
  { A _tmp(b); // Conversion par construction
    f(_tmp);
  }
}
```

Un constructeur ne recevant qu'un seul paramètre est une conversion par construction. Il faut généralement écrire un opérateur d'affectation pour tous les constructeurs n'ayant qu'un seul paramètre.

CHAPITRE 2

CONCEPTION D'IMPLEMENTATION

Vous trouverez dans ce chapitre, des « guides » vous permettant de mieux tirer parti du C++. Lors de la rédaction d'une classe ou d'une méthode, il faut se poser un certain nombre de questions qui devraient en grande partie être résolues lors de la phase de conception. Avant d'entamer la phase de codage, ces guides vous permettront de vérifier une dernière fois la conception détaillée. Pour obtenir une qualité renforcée de votre programme, vous devez pouvoir justifier chacun des choix effectués.

Afin de permettre une lecture partielle de ce chapitre, certains points sont dupliqués lors des différentes étapes. Cela permet de ne consulter que l'étape voulue sans avoir à relire entièrement ce chapitre.

A. REDACTION D'UNE CLASSE

Pour rédiger une classe, nous proposons une liste de question à se poser. Celle-ci vous aidera à chaque étape de développement. La plupart des questions indiquées devraient être résolues lors des phases de « conception » et de « conception détaillée », mais « mieux vaut tard que jamais » ! A l'issue de ce questionnement, votre programme offrira plus de possibilité d'évolution et une facilité d'utilisation accrue.

a. Choix de la classe

- Le rôle de la classe est-il clair ? Avant de rédiger une classe, il faut parfaitement en maîtriser le rôle exact. Ce rôle doit être simple et facilement explicable. Il ne faut pas donner trop d'objectifs à une classe. Sa justification doit être évidente.
- Vérifier que la classe possède un nom clair et explicite.
- Quels en sont les invariants (cf. page 238) ?

b. Attributs

- Commencez par choisir les attributs minimums justifiant l'existence de la classe.
- Par la suite, ajoutez les attributs permettant de l'optimiser. Une méthode constante est un attribut dérivé !
- Éviter au maximum les attributs d'identification d'une instance. Ils empêchent les copies de la classe (*cf.* page 121).
- Traduire l'agrégation par un attribut.
- Traduire l'agrégation en pointeur si l'attribut est virtuel (*cf.* page 51).
- Offrir les services de manipulation de la classe qui utilisent des attributs, mais ne pas offrir d'accès direct aux attributs si cela n'est pas justifié (*cf.* page 42).
- Ne pas déclarer de références comme attribut.
- Éviter les attributs `static`.

c. Relations

- Bien identifier les relations des agrégations. Les accès doivent être différents malgré la même implantation.

- Justifier les sens de parcours d'une relation. Éviter les parcours bidirectionnels.
- Vérifier s'il s'agit d'une relation ou d'une agrégation. Qui doit s'occuper d'effacer l'objet en relation ? S'il s'agit toujours de la classe pointant sur l'objet, alors traduire la relation en agrégation.
- Ajouter les pré- et postconditions sur la cardinalité et sur la réciprocité des relations.

d. Constructeur

- Est-il possible d'avoir un constructeur par défaut ?
- Quels sont les paramètres réellement nécessaires aux constructeurs ?
- Peut-on réduire le nombre de paramètres par des attributs par défauts ?
- Offrir plusieurs constructeurs pour répondre directement aux différents besoins de construction.
- Si l'objet ne possède pas d'attribut identifiant l'instance, offrir le constructeur de copie.
- Si l'objet possède un attribut identifiant l'instance, déclarer le constructeur de copie en `private` (*cf.* page 121).
- Le constructeur de copie doit gérer tous les pointeurs de la classe.
- Appeler les constructeurs :
  1) Des classes héritées si nécessaire.
  2) Des attributs de la classe, dans l'ordre de déclaration.
- Initialiser le maximum d'attribut en dehors des accolades du constructeur (*cf.* page 212).
- Gérer les erreurs apparaissant lors de l'exécution du constructeur par des exceptions.
- Ne pas appeler de méthode `virtual` dans le constructeur.

e. Méthodes

- Offrir une interface cohérente. Bien réfléchir aux différentes possibilités d'utiliser la classe. Imaginer les utilisations de celle-ci différentes de l'exploitation connue pour l'application. Simuler si possible le comportement habituel d'un objet de base.
- Réduire le nombre de paramètres en utilisant les paramètres par défauts.
- Utiliser la surcharge pour des méthodes de sémantiques similaires. Moins il y a de noms différents de méthodes, plus leurs utilisations sont aisées.

- Rédiger toutes les méthodes en constantes par défaut. Ne supprimer l'attribut `const` d'une méthode que si celle-ci modifie l'objet.
- Rédiger des méthodes petites et cohérentes. Déclarer les sous-traitements dans des méthodes `protected` et éventuellement `virtual`.
- Surcharger certains opérateurs par des méthodes.
- Offrir une méthode `clone()` pour tous les objets polymorphiques (*cf.* page 37).

f. Opérateurs

- Offrir les opérateurs permettant de considérer la classe comme un type de base.
- Offrir les opérateurs de test unaire pour vérifier la validité d'une instance (`operator !()` et `operator void*()`).
- Offrir l'ensemble des opérateurs proches entre eux (`operator ==()` et `operator !=()`) et garantir la réflexivité, la commutativité et la transitivité des opérateurs.
- Si l'objet ne possède pas d'identifiant, rédiger l'opérateur d'affectation (*cf.* page 121). Il doit y avoir un opérateur d'affectation pour tous les constructeurs n'ayant qu'un seul paramètre.
- Si l'objet possède un identifiant, déclarer l'opérateur d'affectation en `private`.
- L'opérateur d'affectation doit gérer tous les pointeurs de la classe.
- Ne pas changer la sémantique classique d'un opérateur.
- Doubler certaines méthodes par des opérateurs. Par exemple, les méthodes `add` ou `insert` peuvent être enrichies par l'`operator +=()`.

g. Conversion

- Offrir les conversions permettant de voir l'objet sous différents points de vue (Attention aux instances avec identifiant).
- Déclarer les opérateurs de conversion en `const`.
- Ne pas retourner de pointeur ou de référence non `const` sur un attribut de la classe.

h. Méthodes virtuelles

- Regarder toutes les méthodes et imaginer les modifications nécessaires à y apporter pour dériver la classe dans différents contextes.
- Déclarer en `virtual` les méthodes pouvant être redéfinies par une dérivation.
- Découper une méthode en sous méthodes `virtual` pour pouvoir adapter finement les algorithmes lors de la dérivation.

i. Destructeur

- Le destructeur doit détruire tous les objets que possède la classe : les objets construits dans le constructeur, et ceux fournis par l'utilisateur de la classe.
- Si une seule méthode virtuelle existe, déclarer le destructeur en `virtual`.
- Ne pas appeler de méthode `virtual` dans le destructeur.

j. Droit d'accès

- Choisir judicieusement les droits d'accès de chaque méthode.
- Réduire au maximum le nombre de méthodes `public`. Une méthode `public` vous limite dans vos latitudes de modification du corps de la classe car vous devez garder la même interface qu'auparavant.
- Ne pas déclarer d'attributs `public`.

B. CREATION D'UNE METHODE

Lors de la rédaction d'une méthode, il faut suivre une liste de règles permettant d'améliorer la qualité de celle-ci. Une méthode doit répondre à plusieurs objectifs simultanément :

- la simplicité d'utilisation
- l'évolutivité
- la concision du code

a. Types

- Déclarer par défaut une méthode comme constante. Supprimer cet attribut si cela est vraiment nécessaire.
- Déclarer une méthode en `inline` si le corps de celle-ci peut être judicieusement optimisé par le compilateur, même pour les méthodes virtuelles (*cf.* page 293).
- Déclarer une méthode `static` si elle ne manipule pas d'instances de la classe.

b. Paramètres

- Réduire le nombre de paramètres nécessaires. Utiliser éventuellement un objet à la place d'une collection de paramètres. Exemple, utiliser un paramètre `CDate` à la place de `jour`, `mois` et `année`.
- Utiliser les paramètres par défaut pour simplifier l'interface de la méthode.
- Traduire le passage « par valeur » par un passage par « référence constante ».
- Si un paramètre est un pointeur, signaler dans le commentaire de la méthode si l'élément pointé sera adopté par l'objet, ou s'il est simplement consulté et reste disponible à la fin de celle-ci.
- Si un paramètre n'est pas adopté par l'objet, déclarer le paramètre en référence constante ou non (*cf.* page 111).
- Vérifier en précondition la validité des paramètres et des pointeurs (*cf.* page 238).

c. Variables

- Réduire la durée de vie d'une variable.
- Ne pas utiliser la même variable pour plusieurs sens différents.
- Préférer la création à l'affectation.
- Réduire les copies d'objets.
- Éviter les variables `static`.

d. Corps

- Découper une méthode en sous méthodes `protected` cohérentes, et éventuellement `virtual` pour pouvoir modifier légèrement celle-ci lors d'une dérivation.

- Utiliser l'opérateur de visibilité global (`::`) pour appeler les fonctions du programme ou de la librairie C.
- Libérer toutes les ressources utilisées dans la méthode.

e. Exception

- Toujours prévoir l'interruption de la méthode par une exception, surtout lors de l'appel d'une sous-méthode virtuelle. Libérer toutes les ressources réservées par la méthode lors d'une exception.
- Ne générer d'exceptions que s'il est difficile de retourner un code d'erreur.

f. Retour

- Vérifier les postconditions et les invariants avant de sortir de la méthode (*cf.* page 238).
- Si la méthode ne doit pas retourner d'objet, retourner si possible un code d'erreur.
- Ne jamais retourner une référence sur un paramètre ou une variable de la méthode.
- Si la méthode retourne un objet, minimiser les appels au constructeur de copie et penser à une exception lors du constructeur de copie de l'objet retourné (*cf.* page 90).
- Retourner une référence constante sur un attribut à la place de la valeur de l'attribut (*cf.* page 42).

CHAPITRE 3

PATTERNS DE PROGRAMMATION

Ce chapitre regroupe une collection d'idiomes de codage pour résoudre des situations génériques. Le livre *Design Patterns* [Gamma et al, DP:94] a introduit des « *Patterns* de conception ». Vous trouverez ici regroupés des « *Patterns* de programmation » spécifiques au C++.

Pour ajouter des fonctionnalités génériques qui ne sont pas proposées par le langage, il faut utiliser des *Patterns* de codage. L'identification d'un *Pattern* permet de maîtriser sa traduction. Par exemple, le C++ possède en natif l'utilisation de méthodes virtuelles. Il est possible de rédiger un mécanisme similaire sans le C++ (Voir « Comment ça marche », page 265). Ce concept étant présent dans le langage, vous l'utilisez maintenant abondamment. Lorsque vous vous exprimez, le terme « méthode virtuelle » est clair pour tous. Il fait immédiatement référence au *pattern* correspondant. Les *patterns* de ce chapitre deviendront aussi naturel que le *pattern* : « méthode virtuelle ».

Par une procédure d'induction, l'expérience nous permet de déduire le général du particulier. « *L'induction, c'est le passage de cas particuliers à l'universel* » (Aristote). Les développeurs sont confrontés régulièrement aux mêmes problèmes sous des formes plus ou

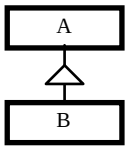
moins variées. Par manque de temps ou de recul, une analyse généraliste du problème n'est pas effectuée. Une solution ponctuelle est trouvée mais sans confrontations aux différentes solutions possibles. L'expérience acquise est alors noyée dans une résolution particulière du problème.

La qualité passe par la traduction de certains concepts à l'aide d'une approche généraliste, robuste et validée. Par la suite, vous ajouterez peut-être de nouveaux « *Patterns* de programmation ».

A. UTILISATION GENERIQUE DE LA CLASSE HERITEE

Il est courant, dans une méthode virtuelle, de rappeler la méthode de la classe de base avant d'ajouter un comportement.

```
class A
{ public:
  virtual void f();
  virtual ~A()
  {}
};

class B : public A
{ public:
  virtual void f();
};

void B::f()
{ A::f(); // Appel de la classe de base
  // ...
}
```



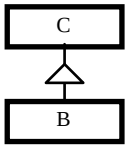
Il serait parfois confortable de pouvoir appeler la méthode de la classe de base et de l'indiquer par un mot clef. Une astuce, proposée par Michael Tiemann [Stroustrup, ARM:94], consiste à ajouter un `typedef` dans la classe.

```
class A
{ public:
  virtual void f();
  virtual ~A()
  {}
};

class B : public A
{ public:
  typedef A Top;
  virtual void f();
};
void B::f()
```

```
{ Top::f();
  // ...
}
```

Si la classe `B` hérite par la suite d'une autre classe que `A`, il n'est plus nécessaire de modifier toutes les méthodes. Il faut juste modifier le `typedef`.

```
class B : public C
{ public :
  typedef C Top;
  // ...
};
```



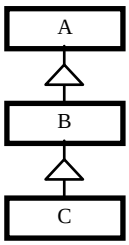
Il n'est pas utile de modifier la méthode `B::f()`. Cela permet d'écrire un code indépendant de la classe de base. Il est courant d'hériter d'une classe ou d'une autre suivant la machine cible.

```
class A :
#if defined(__MSDOS__)
public WndMsdos
{ typedef WndMsdos Top;
#else
public WndUnix
{ typedef WndUnix Top;
#endif
  // ...
};
```

Cette technique permet de simplifier le code des méthodes.

De plus, il faut toujours appeler la méthode de la classe dont on hérite en direct, même si elle n'est disponible que par un héritage supplémentaire (Voir la règle CPP.STY.34, page 162).

```
class A
{ public:
  virtual void f();
  virtual ~A()
  {}
};

class B : public A
{};

class C : public B
{ public:
  virtual void f()
  { B::f();                          // Appel de A::f() car
    // ...                             B::f() n'existe pas !
  }
};
```



L'appel de la version précédente de `f()` dans la version `C::f()` utilise la classe `B` comme scope, alors que cette version n'existe pas. Le compilateur regarde quelle méthode `f()` est visible dans la classe `B`. Il trouve la méthode `A::f()`. Il génère l'appel de celle-ci. Si par la suite, la classe `B` ajoute sa propre version de `f()`, la version de `C` continuerait d'être correcte.

B. COMMENT IMPLANTER LE TYPE `bool` ?

Pour implanter un type `bool`, trois méthodes sont possibles.

a. enum

En déclarant un « `enum bool { false,true };` », on déclare les valeurs possibles et un nouveau type. L'inconvénient est que ce type ne permet pas des écritures comme :

```
bool flag=(x==y);
```

Il faut ajouter une conversion pour que cela fonctionne.

```
bool flag=(bool)(x==y);
```

b. typedef

Une autre solution consiste à utiliser un `typedef`.

```
typedef int bool;
enum {false,true};
```

Dans ce cas, l'écriture précédente fonctionne, même sans conversion.

```
bool flag=(x==y);
```

Par contre, cette déclaration ne fait pas de `bool` un vrai type à part entière, mais plutôt un synonyme de `int`.

Celui-ci ne permet pas de différencier l'appel de deux méthodes surchargées.

```
void fn(bool f);
void fn(int f);
```

Ce que permet pourtant l'`enum`.

c. class

Une troisième solution consiste à déclarer une classe.

```
enum {false,true};
class bool
{ int _bool;
  public:
    bool()                                  { }
    bool(int f) : _bool((f) ? true : false) { }
  bool& operator =(int f)
  { _bool=f ? true : false;
    return *this;
  }
    operator int()                          { return _bool; }
  // ...
};

ostream& operator <<(ostream& o,bool x)
{ return o << ((x) ? "true" : "false"); }
```

Cette version réunit les avantages des deux versions précédentes. Il est possible d'écrire :

```
void f(int x)  { /*...*/ }
void f(bool x) { /*...*/ }

void main()
{ int  a=1;
  int  b=2;
  bool f1=false;
  bool f2=true;
  bool f3=1;
  bool f4=(a==b);

  f1=false;
  f2=true;
  f3=1;
  f4=(a==b);

  f(1);                                     // Appel f(int)
  f(f1);                                    // Appel f(bool)
  if (f1==f2) /*...*/;
}
```

Par contre, certaines écritures ne sont toujours pas correctes. Il faut y ajouter une conversion.

```
f((bool)(a==b));
```

L'initialisation d'une structure avec un objet `bool` ne fonctionne pas.

```
struct
{ int i;
  bool fl;
} Stat={0,false}; // Erreur
```

Cette solution est directement compatible avec la future norme du C++. La conversion deviendra redondante, mais ne sera pas gênante.

Il n'est pas possible d'avoir une solution satisfaisante dans tous les cas avec le langage tel qu'il est défini dans la version 3.

La prochaine norme ANSI/ISO résout ce problème. Un nouveau type a été défini, le type `bool`.

Celui-ci est un type à part entière. Toutes les expressions de test du langage retournent un objet de ce type. Il existe une conversion implicite de `int` vers `bool` et de `bool` vers `int`. Les deux valeurs possibles de cet objet sont `false` et `true`. Toutes les valeurs différentes de `false` sont `true`. Ce type est signé. Cela veut dire qu'avec une conversion implicite, nous avons :

```
void f(unsigned);
void f(int);

void main()
{ bool b=true;

  f(b); // Appel de f(int)
}
```

L'opérateur `++` a été ajouté pour forcer à `true` l'objet. Ce choix particulier a été effectué pour maintenir une portabilité avec les développements précédents et pour faciliter l'introduction de ce nouveau type. Il est clair qu'il n'est pas classique d'utiliser cet opérateur pour forcer à `true`, les compilateurs pouvant d'ailleurs signaler par un `warning`, cette utilisation particulière.

C. RECEVOIR UN PARAMETRE `const char*`

Si vous déclarez une classe recevant un paramètre `const char*`, il faut tenir compte de tous les contextes possibles.

Il existe trois types possibles de `char`. L'un sans attribut, un type signé, et un, non signé. De plus, la nouvelle norme du C++ introduit un objet standard : `string`. Celui-ci ne pos-

sède pas de conversion implicite en `const char*`. Il faut alors, pour le confort de l'utilisateur de votre classe, ajouter des méthodes qui surchargent le service.

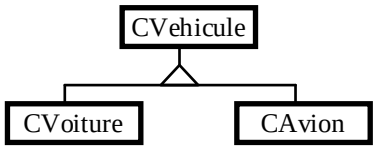
```
class CFile
{
   void open(const char* nom) { /*...*/ }

 // Ajoutez
   void open(const signed char* nom)   { open((const char*)nom); }
   void open(const unsigned char* nom) { open((const char*)nom); }
   void open(const string& nom)        { open(nom.data());       }
};
```

Il est difficile d'offrir le même service avec un constructeur. Il faut dans ce cas appeler explicitement une autre version du constructeur. Il faut utiliser l'opérateur `new` particulier recevant un pointeur en paramètre.

```
inline void* operator new(size_t,void* p) { return p; }
class CFile
{ // ...
    CFile(const char* nom)
    { /*...*/ }
    CFile(const signed* nom)
    { ::new (this) CFile((const char*)nom); }
    CFile(const unsigned* nom)
    { ::new (this) CFile((const char*)nom); }
    CFile(const string& nom)
    { ::new (this) CFile(nom.data()); }
};
```

D. CLONAGE

Il est très fréquent de vouloir construire une copie d'un objet. Le constructeur de copie est là pour cela. Comment obtenir une copie d'un objet dont on ne connaît que la classe de base ? Pour le modèle suivant :



si vous manipulez un pointeur ou une référence de type `CVehicule` et que vous désirez une copie de l'objet référencé, vous ne savez pas si vous devez appeler le constructeur de copie de `CVoiture` ou de `CAvion`.

```
void duplique(const CVehicule& vehicule)
{ CVehicule* dup=new CVoiture((const CVoiture&)vehicule); // ???
             ou new CAvion((const CAvion&)vehicule);     // ???
  //...
  delete dup;
}
```

Pour résoudre cela, il faut ajouter une méthode virtuelle à la classe `CVehicule` qui s'occupera de construire une copie de `this`. Cette méthode sera adaptée pour chaque classe dérivée.

```
class CVehicule
{ //...
  public:
  virtual CVehicule* clone() const=0;
  virtual ~CVehicule() {}
};

class CVoiture : public CVehicule
{ //...
  public:
  virtual CVehicule* clone() const
  { return new CVoiture(*this); }
};

class CAvion : public CVehicule
{ //...
  public:
  virtual CVehicule* clone() const
  { return new CAvion(*this); }
};
```

La fonction `duplique` devient alors :

```
void duplique(const CVehicule& vehicule)
{ CVehicule* dup=vehicule.clone();
  //...
  delete dup;
}
```

Si la fonction `duplique` est un `template`,

```
template <class T>
void duplique(const T& obj)
{ T* dup=obj->clone();
  //...
```

```
  delete dup;
}
```

elle oblige l'objet paramètre `T` à posséder une méthode `clone()`. Pour pouvoir utiliser cette fonction avec tous types d'objets, il faut offrir une fonction `template` gérant le clonage des objets.

```
template <class T>
inline T* clone(const T& x)
{ return new T(x); }
```

Cette fonction utilise le constructeur de copie de la classe paramètre pour cloner l'objet et fonctionne pour tous types d'objets non polymorphes.

```
void main()
{ int a;
  int* pdup=clone(a);
  //...
  delete pdup;
}
```

Les objets polymorphiques comme `CVehicule` doivent la surcharger.

```
inline CVehicule* clone(const CVehicule& x)
{ return x.clone(); }
```

La fonction `duplique()` précédente devenant

```
template <class T>
void duplique(const T& obj)
{ T* dup=clone(obj);
  //...
  delete dup;
}
```

Dans beaucoup de situations il est utile d'avoir cette fonction `clone` (*cf.* « Attribut virtuel », « Durée de vie des objets », « Constructeur de copie et objets polymorphes », ...).

Par exemple, un conteneur d'objets en agrégation contient des objets divers. Ces objets sont détruits lors de la destruction du conteneur. Tous les objets ajoutés au conteneur sont adoptés par celui-ci (Voir « Agrégation », page 16).

```
template <class T>                          // Le paramètre T doit être
class CList                                 // un pointeur ou un objet
{                                           // Simulant un pointeur
  struct CNode
  { T      _data;
    CNode* _next;
    public:
      CNode(T data,CNode* next)
    : _data(data), _next(next) {}
  };
  CNode* _first;
  public:
  CList()                                   // Constructeur
  : _first(NULL)
  { }
  CList(const CList<T>& x);                 // Constructeur de copie

  void adopt(T p)                           // Adoption
  { _first=new CNode(p,_first);
  }
  // ...
  ~CList()                                  // Destructeur
  { CNode* old;
    CNode* cur=_first;
    while (cur!=NULL)
    { delete cur->_data;
      old=cur;
      cur=cur->_next;
      delete old;
    }
  }
};
```

Comment rédiger le constructeur de copie de ce conteneur ? Copier un conteneur d'agrégation revient à copier tous les objets contenus. Le constructeur de copie de la classe `CList` doit dupliquer chacun des objets contenus. Pour cela, il peut utiliser la fonction `template clone().`

```
template <class T>
CList<T>::CList(const CList<T>& x)
: _first(NULL)
{ for (CNode* cur=x._first;cur!=NULL;cur=cur->_next)
    adopt(clone(*cur->_data));
}
```

Il est alors possible d'utiliser ce conteneur d'agrégation pour tous les types d'objets.

```
void main()
{ CList<int*> contEntiers;
  contEntiers.adopt(new int(3));

  CList<CVehicule*> contVehicules;
  contVehicules.adopt(new CVoiture());
```

```
  contVehicules.adopt(new CAvion());
  CList<CVehicule*> contVehiculesClone(contVehicules);
}
```

Le conteneur `contVehiculesClone` possède une copie de la voiture et de l'avion adopté par `contVehicule`. La fonction `clone()` permet de rédiger ce type de service quelque soit la classe de l'objet manipulé.

Toutes les classes polymorphiques devraient posséder une méthode `clone()` virtuelle, et surcharger, pour la classe de base, la fonction `clone()` comme indiqué ci-dessus. La fonction `clone` est une fonction « fabricante » (Voir « Fabricant », page 20).

E. PREVOIR LE `namespace`

La future norme du C++ ajoute un nouveau mot clef : `namespace`. Celui-ci permet d'ajouter un niveau de résolution supplémentaire afin de séparer les différents modules d'une application. L'addition de plusieurs modules peut entraîner des conflits de noms entre les objets. Pour résoudre ce problème, les développeurs ajoutent un préfixe à toutes leurs fonctions pour les identifier indépendamment des autres. Cela entraîne une lourdeur de développement. Pour résoudre ce problème, la future norme permettra d'indiquer dans quel espace de nom sont déclarés les objets. Un module utilisera un espace de nom qui lui est propre, ce qui évitera l'utilisation des préfixes.

En attendant cette version des compilateurs, il est recommandé de placer toutes les variables statiques ou les fonctions globales dans une structure. Celle-ci ne possède pas d'attribut non statique. Elle n'est utilisée que pour ajouter un niveau de résolution. Cela oblige le développeur à préfixer l'appel de ces éléments, mais en utilisant pour cela la syntaxe du C++. Par la suite, il sera facile de supprimer cette classe et d'utiliser le `namespace`.

```
struct Word
{
  static int nbDocument;
  static int trace(const char* str);
};

// ...

void Module() // Utilisation
{
  ++Word::nbDocument;
  Word::trace("Ajoute un document");
}
```

Par la suite, cette classe deviendra :

```
namespace Word
{
  int NbDocument;
  int trace(const char* str);
};

// ...

void Module() // Utilisation
{
  ++Word::NbDocument;
  Word::trace("Ajoute un document");
}
```

L'utilisation reste identique.

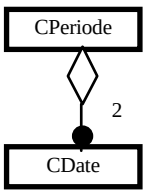
F. ATTRIBUTS ET RELATION

a. L'accès aux attributs

Il arrive fréquemment que l'on désire offrir l'accès à un attribut d'une classe. Plusieurs solutions sont alors possibles. La plus simple consiste à rendre `public` cet attribut. Cette approche est à déconseiller car vous auriez alors des difficultés à modifier votre classe. Si vous désiriez supprimer, par la suite, l'attribut, l'interface de votre classe changerait, et vous devriez réviser les accès à celle-ci. Il semble préférable d'offrir un accès aux attributs à l'aide de méthodes `inline`.

Prenons un exemple d'agrégation simple :

```
class CDate
{ int _jour;
  int _mois;
  int _annee;
  public:
    CDate(int jour,int mois,int annee)
  : _jour(jour), _mois(mois), _annee(annee)
  { }
    CDate(const CDate& x)
  : _jour(x._jour), _mois(x._mois), _annee(x._annee)
  { }
  int jour() const  { return _jour; }
  int mois() const  { return _mois; }
  int annee() const { return _annee; }
  CDate& operator ++()
  { // Ajoute un jour
```



```
    // ...
    return *this;
  }
};

class CPeriode
{ CDate _debut;
  CDate _fin;
  public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
  : _debut(dateDebut), _fin(dateFin) {}

  //...
};
```

La méthode `CDate& CDate::operator++()` n'est pas rédigée complètement. Elle permet d'ajouter un jour à la date. Le changement de mois et d'année devant être pris en compte. Cet opérateur n'est pas du type `const` car il modifie l'objet.

Accès avec get et set

Nous désirons offrir des accès aux attributs `_debut` et `_fin` de `CPeriode`. La première approche consiste à offrir pour ces deux attributs les services `get` et `set`, souvent appelés « accesseurs ».

```
CDate getDebut() const     { return _debut; }
void  setDebut(CDate date) { _debut=date; }
CDate getFin() const       { return _fin; }
void  setFin(CDate date)   { _fin=date; }
```

Les services `getX()` retournent une copie de l'attribut concerné. Les services `setX()` copient dans les attributs un nouvel objet `CDate`. Il est possible de réduire les copies des objets `CDate` nécessaires en modifiant légèrement ces méthodes.

```
const CDate& getDebut() const     { return _debut; }
void  setDebut(const CDate& date) { _debut=date; }
const CDate& getFin() const       { return _fin; }
void  setFin(const CDate& date)   { _fin=date; }
```

Ces nouvelles versions réduisent le nombre de copies nécessaires. Les retours des méthodes `getX()` sont des références sur des constantes. Elles pointent directement sur les attributs de l'objet `CPeriode`. Les paramètres des méthodes `setX()` étant des références sur des constantes, l'objet `CDate` fourni par l'utilisateur de la classe n'est pas copié dans le paramètre de la méthode. Celui-ci aurait ensuite dû être recopié dans l'attribut, et enfin détruit avant de sortir de celle-ci.

L'accès à ces attributs est parfaitement contrôlé par la classe `CPeriode`. Regardons comment l'utilisateur peut modifier un de ceux-ci. Par exemple, cherchons à ajouter un jour à l'attribut `debut` d'une instance `CPeriode`.

```
void main()
{ CPeriode periode(CDate(1,1,1995),CDate(31,1,1995));

  CDate x=periode.getDebut();
  ++x;
  periode.setDebut(x);
}
```

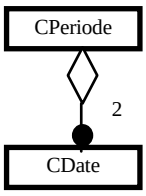
Il faut prendre une copie de l'attribut _debut, puis appeler l'opérateur `++` de cette copie, et enfin demander la mise à jour de l'attribut. Cette mise à jour sera faite par l'opérateur d'assignation de `CDate` appelé dans `setDebut()`. On constate que la manipulation d'un attribut d'une classe avec cet accès, est extrêmement lourd. Imaginez que l'attribut soit un arbre complexe, auquel vous désirez ajouter un élément. Il faudra créer une copie de cet arbre, ajouter l'élément voulu, puis détruire l'arbre de `CPeriode`, et recopier l'arbre temporaire modifié pour enfin détruire l'arbre de la copie.

Cet interface oblige à avoir :

- Une copie via le constructeur de copie.
- La modification de la copie de l'attribut.
- Une affectation qui correspond à un destructeur suivi d'un constructeur de copie.
- Une destruction de la copie.

Regardons comment la classe `CPeriode` peut évoluer. Admettons qu'il existe une classe `CDateEtendu` héritant de `CDate`. Une nouvelle version de `CPeriode` utilise dorénavant deux attributs de type `CDateEtendu`. Cette nouvelle version modifie l'opérateur d'incrémentation pour tenir compte des années bissextiles.

```
class CDateEtendu : public CDate
{ public:
    CDateEtendu(int jour,int mois,int annee)
  : CDate(jour,mois,annee)
  { }
    CDateEtendu(const CDate& x)
  : CDate(x)
  { }
  CDateEtendu& operator ++()
  { // Ajoute un jour en tenant compte des années bisextiles
    return *this;
  }
};

class CPeriode                              // Version 2
```



```
{ CDateEtendu _debut;
  CDateEtendu _fin;
  public:
  // ...
};
```

Plusieurs approches sont possibles pour adapter la nouvelle classe `CPeriode` en maintenant une compatibilité ascendante avec la version précédente. La première consiste à maintenir intégralement l'interface.

```
class CPeriode                              // Version 2.1
{ public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
  : _debut(dateDebut), _fin(dateFin) {}

  const CDate& getDebut() const     { return _debut; }
  void  setDebut(const CDate& date) { _debut=date; }
  const CDate& getFin() const       { return _fin; }
  void  setFin(const CDate& date)   { _fin=date; }
  //...
};
```

L'utilisateur devant manipuler une copie de l'attribut, il manipule une instance de `CDate` et non une instance de `CDateEtendu`. L'appel de l'operateur `++()` ne tiendra pas compte des années bissextiles. Modifions légèrement l'interface, pour tenir compte du nouveau type.

```
class CPeriode                              // Version 2.2
{ public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
  : _debut(dateDebut), _fin(dateFin) {}

  const CDateEtendu& getDebut() const     { return _debut; }
  void  setDebut(const CDateEtendu& date) { _debut=date; }
  const CDateEtendu& getFin() const       { return _fin; }
  void  setFin(const CDateEtendu& date)   { _fin=date; }
  //...
};
```

L'utilisateur aura une erreur lors de l'appel des méthodes `setX()` car celles-ci désirent maintenant une classe de type `CDateEtendu` et non une instance de `CDate`.

```
void main()
{ CPeriode periode(CDate(1,1,1995),CDate(31,1,1995));

  CDate x=periode.getDebut();
  ++x;
  periode.setDebut(x);                      // Erreur !
}
```

L'utilisateur de la classe devra modifier toutes les utilisations de `setX()` pour tenir compte du nouveau type.

```
void main()
{ CPeriode periode(CDate(1,1,1995),CDate(31,1,1995));

  CDateEtendu x=periode.getDebut();
  ++x;
  periode.setDebut(x);
}
```

L'interface à l'aide de `setX()` et `getX()` n'est pas efficace et n'est pas évolutive.

Accès direct

En changeant l'interface, il est possible d'offrir l'accès à l'attribut *in situ*. Retournons une référence non constante sur l'attribut.

```
CDate& debut() { return _debut; }
```

Avec cet accès, il possible de modifier l'attribut directement dans l'objet.

```
++periode.debut();
```

Il faut également offrir un accès à celui-ci en constante.

```
const CDate& debut() const { return _debut; }
```

Si par la suite, la nouvelle version de `CPeriode` utilise la classe `CDateEtendu`, il faut adapter les méthodes d'accès.

```
CDateEtendu& debut()             { return _debut; }
const CDateEtendu& debut() const { return _debut; }
```

L'utilisation directe de l'attribut sera toujours valide.

C'est strictement identique à présenter l'attribut en `public`. La seule différence entre cette approche et l'attribut `public` est qu'il est possible, par la suite, de modifier la méthode `debut()` pour retourner un objet simulant le type `CDate` et offrant une modification du nouvel attribut de la classe `CPeriode`. Si vous modifiez la classe `CPeriode` en supprimant les attributs `CDate`, vous pouvez offrir une interface de la classe, compatible avec l'interface précédente, en retournant un objet simulant `CDate` (Voir « Retour d'objet intermédiaire », page 81). Mais, pour un simple changement de type d'un attribut, c'est très compliqué .

Duplication des services

Si on ne désire pas offrir l'accès direct aux attributs d'un objet, il faut offrir l'interface avec toutes les méthodes des attributs. Dans l'exemple précédent, la classe `CPeriode` doit être rédigée comme suit :

```
class CPeriode                              // Version 2
{ public:
    CPeriode(const CDateEtendu& dateDebut,const CDateEtendu& dateFin)
  : _debut(dateDebut), _fin(dateFin) {}

  void  incDebut() { ++_debut; }
  void  incFin()   { ++_fin; }
  //...
};
```

La classe `CPeriode` doit avoir les méthodes de manipulation pour modifier directement les attributs sans avoir à faire de copies et sans devoir leur fournir un accès `public`. Les méthodes `get`*X*`()` et `set`*X*`()` ne sont pas obligatoires. Elles doivent être ajoutées si l'on désire vraiment offrir ces accès. La manipulation des attributs par la classe est plus efficace. Il faut éviter d'inciter l'utilisateur à copier les objets. Les méthodes `get`*X*`()` seront souvent présentes mais pas les méthodes `set`*X*`()`. Avec cette approche, il est possible de modifier radicalement la classe `CPeriode` sans changer l'interface `public` de celle-ci. Les attributs `CDate` peuvent même disparaître de la classe `CPeriode`.

Accès à une agrégation du type pointeur

Si votre objet possède une agrégation effectuée par un pointeur, il faut offrir une interface compatible avec l'agrégation indépendamment du fait qu'elle soit rédigée avec un pointeur. Supposons que la classe `CPeriode` possède deux pointeurs sur `CDate` à la place de deux attributs.

```
class CPeriode
{ CDate* _debut;
  CDate* _fin;
  public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
  : _debut(new CDate(dateDebut)), _fin(new CDate(dateFin)) {}
    ~CPeriode()
  { delete _debut;
    delete _fin;
  }
  //...
};
```

Il est possible d'offrir des accès à ces attributs en retournant des pointeurs.

```
CDate* getDebut() const { return _debut; }
```

Cette méthode peut être constante car l'objet `CPeriode` n'est pas modifié. Par contre, l'attribut `_debut` peut l'être par l'appelant, ce qui revient, par effet de bord, à modifier l'objet `CPeriode`. Il faut, pour respecter l'agrégation, offrir un accès légèrement différent :

```
const CDate* getDebut() const { return _debut; }
```

Avec cette écriture, la notion d'agrégation est préservée. L'attribut `const` d'une méthode doit indiquer la possibilité de modifier l'objet, quels que soient les moyens d'accès. C'est différent d'une relation. Un objet en relation peut, par principe, être modifié tous seul. Dans ce cas, une méthode d'accès constante peut retourner un pointeur non constant. La classe `CEmploye` ci-dessous possède une *relation* avec une entreprise.

```
class CEmploye
{ CEntreprise* _entreprise;
  public:
  CEntreprise* getEntreprise() const
  { return _entreprise; }
};
```

Plusieurs employés peuvent être en relation avec la même entreprise.

Si l'attribut pointeur doit toujours être présent, la valeur `NULL` étant impossible, il est préférable de retourner une référence sur l'attribut (Voir « Quand et où utiliser les références », page 111).

```
const CDate& getDebut() const { return *_debut; }
```

Le choix de l'implantation de l'agrégation en pointeur ne concerne pas l'utilisateur de la classe qui doit pouvoir utiliser cette dernière comme si l'agrégation était classique, comme si la classe utilisait des attributs.

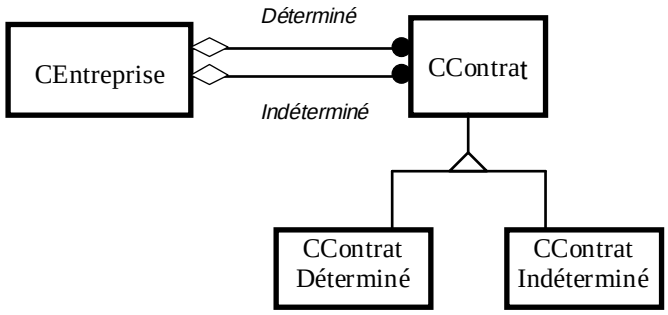
Accès aux conteneurs

Un « conteneur » est un objet regroupant un ensemble d'objet. Un tableau ou une liste chaînée sont des conteneurs. Un conteneur est généralement associé à un « itérateur ». Un itérateur est un curseur permettant de parcourir l'ensemble des éléments d'un conteneur.

Les objets possèdent très souvent des attributs étant des conteneurs. En effet, cela permet de rédiger les relations entre objets. Cela permet également d'écrire une agrégation multiple. Tous les objets contenus appartiennent au conteneur.

Prenons un exemple d'attribut de type conteneurs. Une entreprise emploie des employés. Deux types de contrats sont possibles : les contrats à durée indéterminée, et les contrats à durée déterminée. Admettons que l'on désire avoir deux attributs de type conteneurs pour

l'objet `CEntreprise`, un pour les contrats à durée indéterminée, et un autre pour les contrats à durée déterminée.




```
class CEntreprise
{ CList<CContrat*> _determine;
  CList<CContrat*> _indetermine;
  public:
  //...
};
```

Comment offrir une interface sur ces conteneurs ? Un première approche consiste à offrir un accesseur.

```
const CList<CContrat*>& getContratDetermine() const
{ return _determine; }
const CList<CContrat*>& getContratIndetermine() const
{ return _indetermine; }
```

L'utilisateur de la classe peut alors parcourir les conteneurs à l'aide de l'itérateur approprié. Pour pouvoir modifier par la suite le type de conteneur de la classe, il faut encapsuler les conteneurs dans des `typedef`. Si vous désirez plus tard, modifier le type `CList<T>` par une liste double-chaînée ou un algorithme de H-code, vous ne modifierez pas l'interface de la classe.

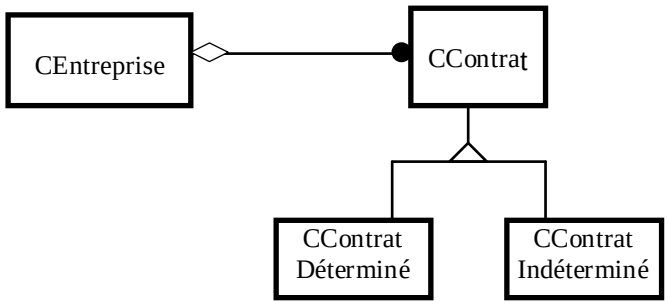
```
class CEntreprise
{ public:
  typedef CList<CContrat*> TContContrat;
  typedef CListIterator<CContrat*> TContratIterator;
  const TContContrat& getDetermine() const
  { return _determine; }
  const TContContrat& getIndetermine() const
  { return _indetermine; }
  private:
  TContContrat _determine;
```

```
  TContContrat _indetermine;
  //...
};
```

L'utilisateur utilisera les types déclarés dans la classe `CEntreprise` pour parcourir les conteneurs.

```
CEntreprise entreprise;

CEntreprise::TContratIterator i(entreprise.getDetermine());
for (i.first();!i.last();++i)
{ //...
}
```

Que se passe-t-il si on désire modifier l'implantation de classes pour n'avoir qu'un seul conteneur, mais cette fois-ci attribué ?



L'interface précédente n'est plus valide. Vous ne pouvez donc pas modifier le corps de votre classe dans ce cas.

Pour pouvoir offrir une interface robuste aux évolutions de la classe, il faut procéder autrement. La classe `CEntreprise` peut être vue comme en conteneur de contrat. Il faut alors rédiger deux itérateurs spécifiques pour cette classe. Un itérateur s'occupera de parcourir les contrats à durée déterminée, et un autre itérateur s'occupera des contrats à durée indéterminée.

```
class CEntreprise
{ CList<CContrat*> _determine;
  CList<CContrat*> _indetermine;
  //...
  friend class CEntrepriseDetermineIterator;
  friend class CEntrepriseIndetermineIterator;
};

class CEntrepriseDetermineIterator : public CListIterator<CContrat*>
```

```
{ public:
    CEntrepriseDetermineIterator(const CEntreprise& entreprise)
  : CListIterator<CContrat*>(entreprise._determine)
  { }
};
class CEntrepriseIndetermineIterator : public CListIterator<CContrat*>
{ public:
    CEntrepriseIndetermineIterator(const CEntreprise& entreprise)
  : CListIterator<CContrat*>(entreprise._indetermine)
  { }
};
```

L'utilisateur peut alors parcourir les conteneurs en utilisant les nouvelles classes.

```
CEntreprise entreprise;

void f()
{ CEntrepriseDetermineIterator i(entreprise);
  for (i.first();!i.last();++i)
  { //...
  }
}
```

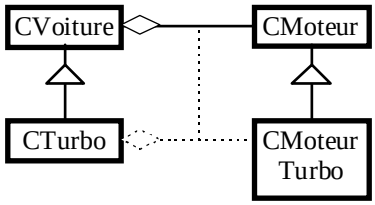
L'interface est plus agréable. Si par la suite, le corps de la classe évolue en ne gardant qu'un seul conteneur, il suffit de modifier les classes itérateurs pour filtrer les éléments parcourus suivant le critère correspondant.

```
class CEntrepriseDetermineIterator : public CListIterator<CContrat*>
{ public:
    CEntrepriseDetermineIterator(const CEntreprise& entreprise)
  : CListIterator<CContrat*>(entreprise._contrats)
  { }
  void first()
  { CListIterator<CContrat*>::first();
    for (;!last() && (data()->duree()!=0);++*this)
      ;
  }
  //...
};
```

Comme pour les attributs simples, il ne faut pas offrir un accès direct à ces éléments, mais offrir les services de manipulation à travers la classe.

b. Attribut virtuel

Il est quelquefois nécessaire de modifier un attribut lors d'un héritage, celui-ci devant être une version dérivée de l'attribut initial. Par exemple, supposons que l'on désire modéliser une voiture, celle-ci étant l'agrégation de plusieurs éléments, un moteur, des roues, etc.



Supposons maintenant que l'on désire dériver de `CVoiture` pour modifier les caractéristiques du moteur. Par exemple, une voiture `CTurbo` possède un moteur ayant de nouvelles caractéristiques. Il faut pouvoir modifier l'attribut `moteur` de `CVoiture` lors de la dérivation. Par analogie avec les méthodes virtuelles qui peuvent être redéfinies dans les classes dérivées, un attribut pouvant être redéfini est appelé « attribut virtuel ».

L'héritage ne permet pas de modifier un attribut de la classe de base car il s'agit d'agrégation. Pour offrir cette possibilité, il faut prévoir dans la classe `CVoiture` la capacité de modifier un attribut. La résolution de ce problème peut être faite comme suit :

```
class CMoteur
{ public:
  virtual int nbChevaux() const
  { return 10; }
  virtual ~CMoteur()
  { }
};

class CVoiture
{ protected:
  CMoteur* _moteur;
    CVoiture(CMoteur* moteur)
  : _moteur(moteur)
  { }
  public:
    CVoiture()
  : _moteur(new CMoteur())
  { }
    ~CVoiture()
  { delete _moteur; }
};
```

Dans un premier temps, la classe `CVoiture` déclare l'attribut `_moteur` comme un pointeur. Celui-ci est valorisé lors du constructeur `public` par une allocation dans le tas. Un constructeur `protected` est déclaré recevant un pointeur de type `CMoteur`. C'est cette version du constructeur que les classes dérivées vont utiliser.

```
class CMoteurTurbo : public CMoteur
{ public:
  virtual int nbCheveaux() const
  { return 25; }
```

```
};

class CTurbo : public CVoiture
{ public:
    CTurbo()
  : CVoiture(new CMoteurTurbo())
  { }
};
```

La voiture `CTurbo` modifie l'attribut `_moteur` de `CVoiture`. Les utilisateurs de ces deux classes ne se soucient pas de cette technique, car les constructeurs `public`s s'utilisent comme si l'attribut était déclaré directement dans l'objet.

Rédiger les constructeurs de copie pour les classes `CVoiture` et `CTurbo` n'est pas aisé. En effet, copier une instance du type `CVoiture` revient à créer une nouvelle instance de l'attribut virtuel. La classe `CTurbo` connaît le type de l'attribut, car c'est elle qui a créé celui-ci. Le constructeur de copie de `CTurbo` n'appellera pas le constructeur de copie de `CVoiture` afin de créer une instance `CMoteurTurbo` à la place de `CMoteur`. Les constructeurs de copie de `CVoiture` et de `CTurbo` se présentent ainsi :

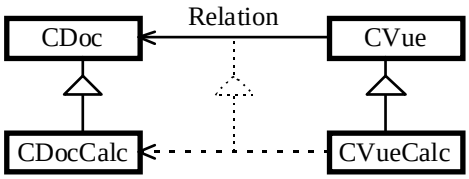
```
  CVoiture::CVoiture(const CVoiture& x)
: _moteur(new CMoteur(*x._moteur))
{ }

  CTurbo::CTurbo(const CTurbo& x)
: CVoiture(new CMoteurTurbo(*(CMoteurTurbo*)x._moteur))
{ }
```

Cette technique permet de résoudre une agrégation virtuelle. Elle peut être rédigée à l'aide des « pointeurs d'agrégations » (Voir « Durée de vie des objets », page 57).

```
class CVoiture
{ protected:
  CPtrHas<CMoteur> _moteur;
    CVoiture(CMoteur* moteur)
  : _moteur(moteur)
  { }
  public:
    CVoiture()
  : _moteur(new CMoteur())
  { }
};
```

c. Relation virtuelle

En C++, il est courant d'avoir un objet ayant une relation virtuelle. Une relation virtuelle est une relation vers un type d'objet, dont contextuellement, on convertit le type vers l'objet dérivé réellement pointé.



Par analogie avec les méthodes virtuelles qui peuvent être redéfinies dans les classes dérivées, une « relation virtuelle » est une relation qui est redéfinie dans les classes dérivées.

L'objet `CDocCalc` hérite de `CDoc`. Ce type de document gère un tableur. L'objet `CVueCalc` hérite de `CVue`. Il permet d'afficher une vue du tableur. Une vue possède un pointeur sur un objet `CDoc`. `CVueCalc` est en relation avec un document du type `CDocCalc`. La relation entre les vues et les documents est héritée par la classe `Vue`. Contextuellement, un objet `CVueCalc` sait que cette relation est du type `CDocCalc`.

```
class CDoc
{ public:
  const char* nom();
};

class CDocCalc : public CDoc
{ public:
  void calcul();
};

class CVue
{ protected:
  CDoc* _useDoc;
};

class CVueCalc : public CVue
{ public:
  void affiche();
};
```

La méthode `CVueCalc::affiche()` désire utiliser la méthode `CDocCalc::calcul()` avant de rafraîchir l'écran. Pour cela, elle utilise le pointeur `_useDoc` hérité de `CVue`. Celui-ci est du type `CDoc`. La méthode doit convertir ce pointeur pour pouvoir accéder à la méthode `calcul()`.

```
void CVueCalc::affiche()
{ ((CDocCalc*)_useDoc)->calcul();
  // ...
}
```

Contextuellement, le pointeur `_useDoc` doit toujours être vu comme un pointeur `CDocCalc` lorsqu'il est utilisé par la classe `CVueCalc`. Ceci est une relation virtuelle. Le langage ne permet pas directement d'offrir une modification d'un type suivant le contexte d'utilisation. Un évolution du C++ pourrait par exemple accepter une écriture du type :

```
// Ceci n'est pas du C++
class CVue
{ protected:
  virtual CDoc* _useDoc;
};

class CVueCalc : public CVue
{ protected
  virtual CDocCalc* _useDoc; // Change le type de l'attribut
  public:
  void affiche();
};
```

Pour éviter d'avoir des conversions dans toutes les méthodes de `CVueCalc`, il est intéressant d'écrire une méthode s'occupant de celles-ci.

```
class CVueCalc : public CVue
{ // ...
  CDocCalc* useDoc() const
  { return (CDocCalc*)_useDoc; }
  public:
  void affiche()
  { useDoc()->calcul();
    // ...
  }
};
```

Les méthodes de `CVueCalc` doivent utiliser la méthode `useDoc()` pour utiliser le pointeur. La conversion est présente à un seul endroit du programme. Si par la suite, la relation doit changer, il ne faut modifier qu'une ligne et recompiler.

L'utilisation des relations virtuelles est très fréquente. L'écriture précédente simplifie son utilisation. Cette écriture part de l'hypothèse que contextuellement, le pointeur de base pointe sur un objet dérivé dont on connaît le type. C'est une hypothèse qui ne se concrétise pas forcément à l'exécution. Supposez qu'un objet `CVueCalc` a une relation vers un objet `CDoc` à la place d'un objet `CDocCalc`. Cela entraînera des erreurs difficiles à localiser. Les méthodes seront appelées comme si l'objet était du type `CDocCalc`. Celles-ci modifieront des données en dehors de l'objet pointé. La nouvelle norme du C++ permet de vérifier qu'une conversion d'un pointeur sur un type de base vers un type dérivé est correcte à l'exécution. Pour cela, il faut utiliser la conversion à l'aide de `dynamic_cast<T>`. Celui-ci vérifie à l'exécution que la conversion est valide.

```
CCalc* useDoc() const
{ assert(dynamic_cast<CDocCalc*>_useDoc!=NULL);
```

```
  return static_cast<CDocCalc*>_useDoc;
}
```

La conversion à l'aide de `static_cast<T>` vérifie à la compilation que celle-ci est valide. Elle est plus rapide mais ne teste pas le pointeur. La classe `CVueCalc` ne devrait jamais utiliser directement le pointeur `_useDoc` car elle l'utiliserait comme un pointeur de type `CDoc`. Pour cela, il faudrait déclarer celui-ci en `private`. Mais, dans ce cas, la méthode de conversion ne peut plus être écrite, car elle ne peut plus accéder au pointeur.

Les macros suivantes permettent de généraliser la résolution de ce besoin.

```
#define VREF(name) _vref_ ## name
#define DCL_VREF(Type,name)                          \
  Type VREF(name);                                   \
  Type name () const { return VREF(name); }          \
  Type& name() { return VREF(name); }

#define USE_VREF(Type,name)                          \
  Type name() const                                  \
  { assert(dynamic_cast<Type>VREF(name)!=NULL);      \
    return static_cast<Type>VREF(name);              \
  }                                                  \
  Type& name()                                       \
  { assert(dynamic_cast<Type>VREF(name)!=NULL);      \
    return static_cast<Type&>VREF(name);             \
  }
```

Le nom de l'attribut de relation est *camouflé* à l'aide d'un préfixe « `_vref_` » afin de réduire le risque d'utilisation de ce pointeur directement.

Si votre compilateur n'offre pas encore les conversions à l'aide de `static_cast` et `dynamic_cast` utilisez les conversions classiques.

L'utilisation est simplifiée. La classe de base déclare la relation virtuelle à l'aide de la macro `DCL_VREF()` et les classes dérivées utilisent `USE_VREF()`.

```
class CVue
{ protected:
  DCL_VREF(CDoc*,useDoc)
  const char* nomDoc() const
  { return useDoc()->nom(); } // Pt Doc*
};

class CVueCalc : public CVue
{ public:
  USE_VREF(CDocCalc*,useDoc)
  void affiche();
};
void CVueCalc::affiche()
{ useDoc()->calcul();         // Pt CDocCalc
```

```
  // ...
}
```

La méthode `useDoc()` s'utilise comme un attribut classique. Ce n'est pas un accesseur, mais un synonyme de l'attribut.

G. GESTION MEMOIRE

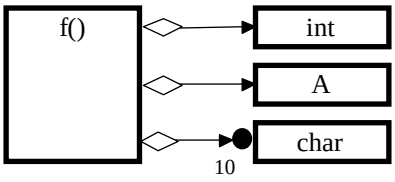
a. Durée de vie des objets

Tout objet C++ possède un constructeur et un destructeur. Le destructeur est appelé lors de la fin de vie de l'objet. Le C++ gère les durées de vies des objets présents dans les fonctions et globaux à l'application. Les zones mémoires demandées dans le tas doivent être détruites par le programme qui les a demandées.

Un objet local, déclaré dans une fonction, sera détruit lorsque celle-ci sera terminée. Le destructeur correspondant sera appelé. On peut considérer les variables créées dans les fonctions comme des agrégations de celles-ci. Par exemple, la fonction suivante :

```
void f()
{ int i;
  A   a;
  char buf[10];
  // Traitement...
}
```

peut être schématisée comme ci-dessous :



Sortir de la fonction permet de détruire les objets de celle-ci. En quelque sorte, appeler une fonction équivaut à créer une *instance d'exécution* de celle-ci. Lors de la destruction d'une instance d'exécution, les attributs de l'instance d'exécution (les variables locales) sont détruits. Il est possible de traduire cette fonction par une classe équivalente :

```
class f
{ int i;
  A   a;
  char buf[10];
  public:
  f()
  { // Traitement sans variable locale
  }
};
```

et l'appel de celle-ci par :

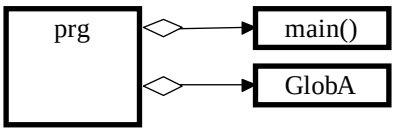
```
{ delete new f();                          // Oupps !!!
}
```

L'appel d'une fonction crée une instance d'exécution de la classe `f`, et le retour de la fonction détruit cette instance (`delete`). Il n'est pas nécessaire d'avoir des variables déclarées dans le constructeur de la classe `f`, car chaque instance de celle-ci possède les attributs nécessaires. Un appel récursif peut être vu comme la création de plusieurs instances d'exécution de la même fonction.

Les objets globaux à l'application sont créés avant l'appel du `main` et détruits après celui-ci.

```
A GlobA;
void main()
{ //...
}
```

se schématise comme suit :



Le compilateur garantit qu'il appellera autant de destructeurs que de constructeurs. L'instance du programme sera détruite après que celui-ci aura libéré toutes les ressources nécessaires à son exécution. En quelque sorte, les variables globales sont les attributs de l'*instance du programme*. Les fonctions et les méthodes statiques étant les méthodes de

l'instance du programme. On pourrait imaginer une extension du langage pour indiquer qu'une fonction ne modifie pas de variables globales et qu'elle n'a pas d'effet de bord.

```
void f() ::const // Ce n'est pas du C++
{ //... Traitement sans modification de variables globales
}
```

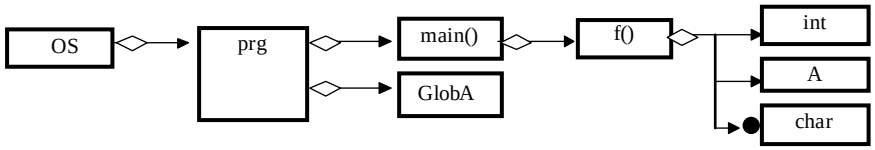
L'opérateur de résolution est ajouté à l'attribut `const` pour lever l'ambiguïté entre l'attribut `const` de la classe et l'attribut `const` global.

```
class A
{ void f() const ::const // Ce n'est pas du C++
  {
  }
};
```

La méthode `f()` de la classe `A` ne modifie pas les attributs de l'instance `this` courante, et ne modifie pas de variables globales.

Les paramètres passés à la fonction `main` ont également une durée de vie gérée par le compilateur. En effet, avant d'appeler la fonction `main`, le programme est exécuté par une fonction de démarrage souvent appelée `crt`. Celle-ci se chargera de traduire les informations fournies par le système d'exploitation et de les convertir dans un formalisme compréhensible par le programme. Ainsi, la ligne de commande et les variables d'environnement sont traduites par la fonction de démarrage pour pouvoir servir de paramètres à la fonction `main`.

Qui s'occupe de la durée de vie des paramètres que fournit le système d'exploitations ? Eh bien, il s'agit du système lui-même. En général, ces informations sont stockées dans la zone mémoire réservée au programme exécuté. Celui-ci informera le système d'exploitation de sa fin, qui s'occupera alors de libérer les zones mémoires qui étaient nécessaires à l'application. Les paramètres complémentaires seront ainsi détruits.



Comme on le voit avec cette description détaillée des rôles de chacun, chaque objet possède un responsable qui s'occupera de le détruire. L'ultime responsable de la gestion des ressources étant le système d'exploitation. Le C++ permet au développeur de gérer plus facilement que le C les allocations mémoires. En effet, si une zone mémoire est demandée pour un objet, le destructeur de celui-ci se chargera de la détruire. Toutes les zones mémoi-

res appartenant à un objet doivent être détruites par celui-ci. Une zone mémoire appartenant à un objet est une *agrégation*. Une agrégation est une relation particulière qui lie la durée de vie de l'objet en relation à celle de l'objet possédant cette relation (Voir « Agrégation », page 16). Si un objet est créé dans une fonction, à la fin de celle-ci l'objet sera détruit et les zones mémoires nécessaires à l'objet également. Une agrégation est également appelé « pointeur *Has* ».

Le C++ gère la durée de vie des objets globaux ou locaux aux fonctions. Comment gérer les durées de vie des objets alloués dans le tas par une fonction ? Un pointeur ne déclare qu'une *relation* avec un objet. Lors de la destruction d'un pointeur, l'objet pointé n'est pas détruit ! Le C++ ne possédant pas de « ramasse miettes », c'est au développeur de gérer cela. Nous allons voir que la difficulté de la gestion mémoire en C++ est essentiellement due à une sémantique ambiguë du rôle des pointeurs. Une fois leur rôle parfaitement identifié, il est possible de rédiger des outils facilitant énormément cette gestion.

Le C++ possède en effet, une ambiguïté très importante dans la sémantique des pointeurs. Un pointeur peut avoir quatre sens différents.

1. Il peut représenter une relation, c'est-à-dire que l'objet pointé n'est pas dépendant de la durée de vie du pointeur.
2. Il peut représenter une relation sur un tableau, c'est-à-dire que l'objet pointé peut être manipulé comme un tableau et sa durée de vie ne dépend pas de l'existence du pointeur. Le pointeur peut se déplacer dans le tableau.
3. Il peut représenter une agrégation, c'est-à-dire que l'objet pointé est dépendant de la durée de vie du pointeur. Le développeur ne doit pas oublier de demander la destruction de l'objet pointé avant de détruire le pointeur.
4. Enfin, il peut représenter un tableau en agrégation, c'est-à-dire que le tableau pointé est dépendant de la durée de vie du pointeur. Perdre la valeur du pointeur empêche de détruire le tableau.

L'ambiguïté sur ces quatre sens est source de beaucoup d'erreurs. Par exemple, le compilateur ne signalera pas une écriture erronée du type :

```
void f()
{ char a,b='b';
  strcpy(&a,&b);
}
```

car il ne fait pas la différence entre l'adresse d'un objet et un tableau.

Pointeur d'agrégation

Les développeurs utilisent souvent des pointeurs pour allouer de la mémoire dans une fonction et ils n'oublient généralement pas de la détruire. Si le développeur demande de la mémoire dans le tas, c'est généralement qu'il ne peut pas la demander dans la pile du programme. Dans ce contexte cette mémoire sert souvent uniquement à la fonction. C'est elle qui est propriétaire de l'allocation. Quelque soit la cause de sortie de la fonction, cette ressource doit être libérée. Pour ajouter un sens au pointeur et offrir une libération automatique quelques soient les causes de sortie de la fonction, il peut être utile de rédiger un objet représentant la durée de vie de l'objet pointé. La destruction de cet objet entraînera la destruction de l'objet pointé.

CPtrAggr

```
template <class T>
class CPtrHas
{ T* _pt;
  public:
    CPtrHas(T* pt=NULL)
  : _pt(pt) {}
    ~CPtrHas()
  { delete _pt; }
  //...
};
```

Cet objet sera détruit automatiquement lors de la fin de la fonction. Il faut ajouter à cet objet les opérateurs permettant de le manipuler comme un pointeur classique.

```
template <class T>
class CPtrHas
{ T* _pt;
  public:
    CPtrHas(T* pt=NULL)
  : _pt(pt) {}
    ~CPtrHas()
  { delete _pt; }
  T* operator ->() const
  { assert(_pt!=NULL);
    return _pt;
  }
  T& operator  *() const
  { return *operator->(); }
    operator T* () const { return _pt; }
  private:
  T& operator [](int index);                // Non implanté
};
```

Il ne faut pas permettre d'utiliser l'objet représentant un pointeur d'agrégation sur un objet comme un pointeur sur un tableau. C'est pour cela que l'`operator []()` n'est pas disponible.

```
{ CPtrHas<A> pa=new A(1);
  pa[1]=A(2); // Erreur à la compilation
}
```

De plus, l'`operator ->()` vérifie que le pointeur n'a pas la valeur `NULL`.

Il est en revanche utile de demander à un objet représentant un pointeur d'agrégation de libérer sa responsabilité sur la durée de vie du pointeur. Pour cela on ajoute une méthode particulière.

```
T* release()
{ T* rc=_pt;
  _pt=NULL;
  return rc;
}
```

Elle permet de transmettre la responsabilité de la durée de vie comme suit :

```
{ CPtrHas<A> pa=new A();
  f(pa.release()); // f() adopte l'objet
}
```

Maintenant posons la question de l'opérateur d'affectation. Quelle sémantique mettre à cet opérateur ? Changer la valeur de ce pointeur entraîne la perte de l'adresse de l'objet précédemment alloué. Or, cet objet représente la durée de vie de l'objet dans le tas. La perte du pointeur entraîne donc la destruction de l'objet pointé.

Il existe trois sens possibles à l'opérateur d'affectation.

1. Affectation impossible sans détachement.
2. Affection transférant la responsabilité de l'objet pointé.
3. Affectation dupliquant l'objet pointé.

- Dans la première hypothèse, il faut rédiger l'opérateur d'affectation en `private`.

```
private:
CPtrHas<T>& operator =(const CPtrHas<T>& x); // Non implanté
```

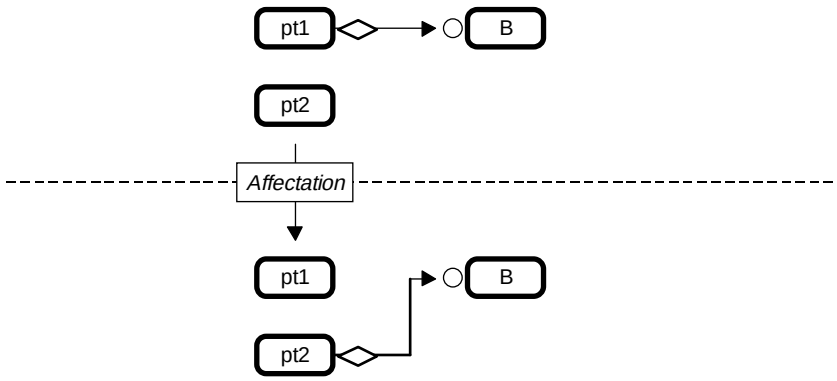
- Dans la deuxième hypothèse, l'affectation doit commencer par détruire l'objet possédé avant d'adopter le pointeur (L'objet `auto_ptr<T>` de la librairie standard ANSI/ISO C++ utilise une sémantique proche).

```
CPtrHas<T>& operator =(T* pt)
{ delete _pt;
  _pt=pt;
  return *this;
}
```

Un objet ne pouvant appartenir qu'à un seul autre objet, modifier la valeur du pointeur indique que l'on désire perdre la valeur précédente. Pour détruire explicitement l'objet possédé il suffit d'effectuer une affectation avec la valeur `NULL`.

```
{ CPtrHas<A> pa=new A();
  pa=NULL; // Destruction de l'objet agrégé
}
```

Une affectation entre deux pointeurs de ce type va transmettre d'un pointeur à l'autre la responsabilité de la durée de vie de l'objet pointé. Il faut, dans ce cas, rédiger un opérateur d'affectation spécial s'occupant de cela.




```
CPtrHas<T>& operator =(CPtrHas<T>& x)
{ delete _pt;
  _pt=x._pt;
  x._pt=NULL;
  return *this;
}
```

Une écriture comme la suivante :

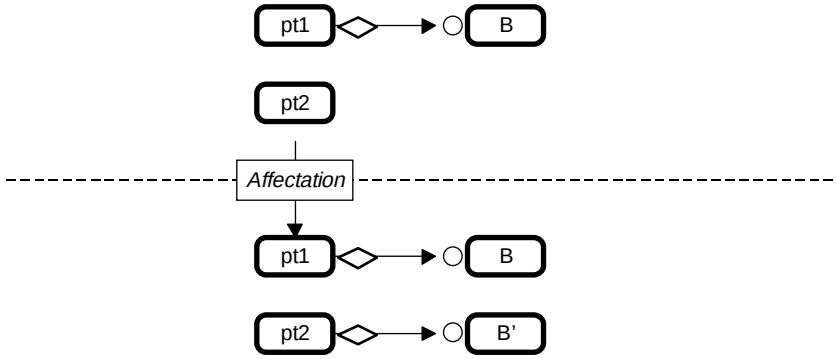
```
{ CPtrHas<A> p1=new A(1);
  CPtrHas<A> p2=new A(2);
  p1=p2;
}
```

entraîne la destruction de l'objet `A(1)` et la récupération de l'adresse de `A(2)` par `p1`. Le pointeur `p2` devient `NULL`. Il faut rédiger de même le constructeur de copie.

```
CPtrHas(CPtrHas<T>& x)
{ _pt=x._pt;
```

```
  x._pt=NULL;
}
```

- Dans la troisième hypothèse, l'affectation va créer à l'aide du constructeur de copie, un duplicata de l'objet agrégé.



Pour des objets polymorphiques, il faut ajouter une méthode `clone` à l'objet agrégé (Voir « Clonage », page 42). Tous les objets ayant une méthode virtuelle doivent avoir cette méthode. Dans ce cas, l'affectation et le constructeur de copie se rédigent comme suit :

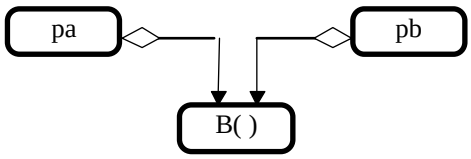
```
CPtrHas(const CPtrHas<T>& x)
: _pt((x._pt!=NULL) ? (T*)clone(*x._pt) : NULL)
{ }
CPtrHas<T>& operator =(const CPtrHas<T>& x)
{ _pt=(x._pt!=NULL) ? (T*)clone(*x._pt) : NULL;
  return *this;
}
```

Cette dernière version est la plus souple. En effet, un pointeur d'agrégation est un représentant de l'objet. Il est assimilable à l'objet. Copier ce représentant est assimilable à copier l'objet lui-même.

Il subsiste néanmoins un problème. Comme nous l'avons vu précédemment, un objet ne peut appartenir qu'à un seul autre objet. C'est pour cela que le constructeur de copie d'un pointeur d'agrégation effectue un `clone()` de l'objet agrégé. Que ce passe-t-il avec une écriture comme celle-ci ?

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

void main()
{ CPtrHas<B> pb=new B();
  CPtrHas<A> pa=pb;                          // Erreur
}
```

L'objet `pa` doit être initialisé à l'aide d'un pointeur sur un objet `A`. Pour cela, le compilateur utilise l'opérateur de conversion de `pb` pour convertir `pb` en relation sur un `B`, celui-ci étant donné au constructeur de `pa`. Les deux pointeurs `pa` et `pb` possèdent alors le même objet !



Pour éviter cette erreur, il faut interdire ce type d'écriture. Les classes `CPtrHas<T>` générées par le compilateur vont hériter d'une même classe. Celle-ci permettra d'interdire l'écriture ci dessus.

Seule une modification du compilateur permettra d'avoir une sémantique de clonage lors de l'affectation entre pointeurs d'agrégations. Pour le moment, l'utilisateur doit modifier son code pour cloner l'objet, ou pour transmettre la responsabilité de l'objet pointé.

```
void main()
{ CPtrHas<B> pb=new B();
  CPtrHas<A> pa=clone(*pb);                  // Duplique l'objet
  pa=pb.release();                           // Transfert la resp.
}
```

Maintenant, quel sens donner à l'`operator ==()` ? Un objet ne peut appartenir qu'à un seul pointeur d'agrégation. Deux pointeurs de ce type ne peuvent pas posséder le même objet. L'égalité entre deux pointeurs d'agrégations n'est jamais vraie. On peut alors envisager de comparer les objets possédés.

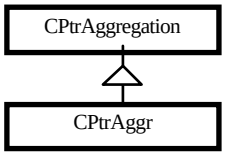
```
bool operator ==(const CPtrHas<T>& p)
{ return *this==*p; }
```

Cela entraîne que : `(*p1==*p2)` est équivalent à `(p1==p2)`.

Avec cet opérateur, il est possible d'écrire :

```
void main()
{ CPtrHas<A> a1=new A();
  CPtrHas<A> a2=new A();
  if (*a1==*a2) ...
  ou
  if (a1==a2) ...
}
```

Voici le code complet de cette classe :




```
class CPtrAggregation {};
template <class T>
class CPtrHas : public CPtrAggregation
{ T* _pt;
  public:
    CPtrHas(T* pt=NULL)
  : _pt(pt) {}
  CPtrHas(const CPtrHas<T>& x)
  : _pt((x._pt!=NULL) ? (T*)clone(*x._pt) : NULL)
  { }
   ~CPtrHas()
  { delete pt; }
  T* operator ->() const
  { assert(pt!=NULL);
    return _pt;
  }
  T& operator  *() const
  { return *operator->(); }
    operator T* () const
  { return _pt; }
  CPtrHas<T>& operator =(T* pt)
  { delete _pt;
    _pt=pt;
    return *this;
  }
  CPtrHas<T>& operator =(const CPtrHas<T>& x)
  { _pt=(x._pt!=NULL) ? clone(*x._pt) : NULL;
    return *this;
  }
  T* release()
  { T* rc=_pt;
    _pt=NULL;
    return rc;
  }
  bool operator ==(const CPtrHas<T>& p)
  { return *this==*p; }
  bool operator !=(const CPtrHas<T>& p)
  { return *this!=*p; }
  protected:
  void operator[](size_t);                  // Non implementé
  void operator =(const CPtrAggregation&);  // Non implementé
```

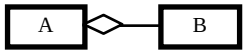
```
  CPtrHas(const CPtrAggregation&);          // Non implementé
};
```

Si vous utilisez cet objet, vous maîtriserez mieux le sens de vos pointeurs. Plus les objets manipulés sont chargés de sens, plus vous maîtriserez votre programme. De plus, cet objet est compatible avec les exceptions. En effet, si une instance de ce pointeur est détruite lors de la remontée de la pile, suite à un `throw`, la mémoire associée sera libérée. Pour pouvoir utiliser les exceptions correctement, il faut justement utiliser cet objet. Ce n'est pas un hasard. Une exception appelle la destruction d'une instance d'exécution de la fonction. Le programme sort du contexte de celle-ci. Il faut alors détruire tous les objets en agrégation de la fonction.

```
void f()
{ CPtrHas<A> buf=new A(1);
  g(); // Fonction generant eventuellement une exception
}
```

De plus, cet outil entraîne une maîtrise parfaite du rôle de chaque pointeur et est directement compatible avec les exceptions. Il faut utiliser ces pointeurs à la place de tous pointeurs représentant une agrégation. Il peut être présent comme attribut d'une classe ce qui facilite la rédaction du destructeur.

```
class A
{ CPtrHas<B> pb;
  public:
    A() : pb(new B()) {}
};
```



Il n'est pas nécessaire de rédiger un destructeur particulier. Le destructeur par défaut est suffisant. L'opérateur d'affectation par défaut est également correctement généré par le compilateur.

Pour retourner un pointeur d'agrégation lors d'une méthode fabricante, rien de plus simple.

```
CPtrHas<B> fabriquant()
{ CPtrHas<B> pb=new B();
  //...
  return pb;
}
```

Les compilateurs modernes ne clonent pas l'objet `B()` car ils détectent que la variable locale `pb` est la variable servant de valeur de retour. Ils génèrent un code manipulant directement la variable de retour, sans la création réelle de la variable locale `pb`. L'utilisateur de cette fonction doit attendre un pointeur d'agrégation.

```
void main()
{ CPtrHas<B> pb=fabriquant();
  //...
}
```

L'objet `B` créé sur le tas n'est jamais cloné lors de cette écriture. Attention, il ne faut surtout pas recevoir un pointeur sur un `B`, car cela représente une relation.

```
void main()
{ B* pb=fabriquant();                    // Erreur
}
```

Le compilateur construit un objet temporaire de type `CPtrHas<B>` pour récupérer l'objet construit par la fonction `fabriquant()`, puis il appelle l'opérateur de conversion d'un pointeur d'agrégation vers un pointeur de relation, et enfin détruit le pointeur d'agrégation. A partir de ce moment, l'objet agrégé est détruit. La relation n'est plus valide.

```
void main()
{ B* pb;
  { CPtrHas<B> _tmp=fabriquant();
    pb=_tmp.operator B*();
    _tmp.CPtrHas<B>::~CPtrHas();         // Efface agrégation
  }
  // pb invalide
}
```

Si vous désirez recevoir une agrégation dans un pointeur de relation, il faut utiliser la méthode `release()`.

```
void main()
{ B* pb=fabriquant().release();
  // ...
  delete pb;
}
```

Il n'est pas conseillé de faire ainsi. Il est préférable de maintenir le sens de l'agrégation du pointeur. La méthode `release()` devrait toujours être appelée pour valoriser un autre pointeur d'agrégation.

Pour recevoir en paramètre un pointeur d'agrégation à consulter, il faut recevoir une relation,

```
void f(const A* p)
{ //...
}
```

afin de ne pas dupliquer l'objet en agrégation.

Il est à noter que la norme ANSI/ISO du C++ indique que si le type paramétré d'un `template` est incompatible avec l'`operator ->()` celui-ci doit être ignoré. Cela permet d'utiliser ce pointeur avec des types de base.

```
CPtrHas<int> pInt=new int(3);
```

Cet objet est également très pratique avec les *Standard Template Library* (STL). Un conteneur d'objets polymorphes se déclare ainsi :

```
vector<CPtrHas<A> > vector;
```

Le constructeur de copies du conteneur, copie tous les objets. Le destructeur détruit les objets contenus.

Ce pointeur d'agrégation permet généralement de se passer d'utiliser l'instruction `delete` du compilateur. Il ne s'agit plus dorénavant, de gérer la mémoire par des allocations et des destructions, mais de demander des ressources au système pour les donner à un objet. Un outil équivalent devrait être présent dans la syntaxe du C++. Énormément d'erreurs seraient ainsi évitées. Cet objet fait le lien entre les objets présents dans la pile et les objets présents dans le tas.

Pointeur de tableau en agrégation

Pour un pointeur de tableau en agrégation, il faut utiliser la classe suivante :

CPtrArrayAggr

```
template <class T>
class CPtrHasArray
{ T* _pt;
  public:
    CPtrHasArray(T* pt=NULL)
  : _pt(pt) {}
    ~CPtrHasArray()
  { delete [] pt; }
  T& operator [](int i) const
  { return _pt[i];
  }
    operator T* () const { return _pt; }
  CPtrHasArray(<T>& operator =(T* pt)
  { delete [] _pt;
    _pt=pt;
  }
  T* release()
  { T* rc=_pt;
    _pt=NULL;
    return rc;
  }
  private:
  CPtrHasArray(const CPtrHasArray<T>& x);
```

```
  CPtrHasArray<T>& operator =(const CPtrHasArray<T>& x);
};
```

Il n'est pas possible de dupliquer un tableau car le C++ ne donne pas suffisamment d'informations pour cela. Il n'existe pas de moyen de connaître la taille d'un tableau. Le constructeur de copie et l'affectation sont alors déclarés en `private`. Utilisez cet objet pour tous types de tableaux demandés dans le tas.

```
void main()
{ CPtrHasArray<A>   arrayA=new A[10];
  CPtrHasArray<int> arrayInt=new int[15];
  // ...
}
```

Un tableau de caractères en agrégation doit utiliser ce type de pointeur.

```
const char initchar[10]={'0','1','2','3','4','5','6','7','8','9'};
void main()
{ CPtrHasArray<char> buf=new char[10];
  memcpy(buf,initchar,sizeof(initchar));
  // ...
}
```

Il ne faut pas appeler `delete [](`). Le pointeur s'en occupe. Pour une chaîne de caractères, utiliser l'objet `string` présent dans la librairie tous les compilateurs (à quelques variantes près). Cet objet représente un pointeur d'agrégation spécialisé dans les tableaux de caractères. La taille du tableau se déduit de la longueur de la chaîne. Le constructeur de copie et l'affectation peuvent alors être rédigés pour ce contexte particulier.

Conclusion

Avec ces deux outils, une partie non négligeable des ambiguïtés sur la sémantique des pointeurs est levée.

Sémantique	Codage
Relation	`T*`
Relation d'un tableau	`T[]`
Agrégation	`CPtrHas<T>`
Agrégation d'un tableau	`CPtrHasArray<T>`

Il ne reste que l'ambiguïté entre une relation vers un objet et une relation vers un tableau. Les traitements par défaut du compilateur sont alors toujours corrects, et les instructions `delete` et `delete []` ne sont pratiquement plus nécessaires !

b. Contrôler la localisation d'un objet

Objet interdit de tas

Le *tas* est la zone mémoire disponible pour l'application qui est allouée à la demande. L'opérateur `::new` et la fonction `malloc()` demande de la mémoire dans le tas. Pour interdire l'allocation d'un objet dans le tas, il suffit d'interdire l'utilisation des opérateurs d'allocation de l'objet.

```
class CObj
{ private:
  void* operator new(size_t);
  public:
  //...
};
```

Objet présent uniquement dans le tas

Si vous désirez qu'un objet ne soit présent que dans le tas, c'est-à-dire qu'il ne puisse être construit qu'avec l'opérateur `new`, il existe plusieurs approches. Premièrement, il est possible de déclarer les constructeurs de l'objet en `protected`, et de fournir les méthodes statiques de la classe permettant de construire l'objet.

```
class CObj
{
  // Ctr protected
  protected:
    CObj();
    CObj(int x);
    CObj(const CObj& x);
  public:
  static CObj* New()              { return new CObj(); }
  static CObj* New(int x)         { return new CObj(x); }
  static CObj* New(const CObj& x) { return new CObj(x); }
};
```

On remarque que cette approche fonctionne, mais qu'il faut déclarer autant de méthodes statiques qu'il y a de constructeurs.

Il peut exister plusieurs constructeurs pour la même classe. Par contre, il ne peut exister qu'un seul destructeur. Utilisons cette caractéristique pour simplifier la résolution de ce problème.

```
class CObj
{ public:
    CObj();
    CObj(int x);
```

```
    CObj(const CObj& x);
  friend void Delete(CObj* x) { delete x; }
  protected:
    ~CObj();
};
```

Avec cette écriture, l'utilisateur n'est pas empêché de construire un objet local, mais lors de la destruction de celui-ci, le compilateur indique qu'il ne peut pas accéder au destructeur de l'objet.

```
{ CObj a(3);
  // ...
} // CObj::~CObj protected !
```

L'utilisateur doit appeler la fonction `Delete()` pour effacer l'objet, et par là même, construire l'objet avec un `new` classique.

```
{ CObj* p=new CObj(3);
  // ...
  Delete(p);
}
```

Pour régler un problème en C++, il faut parfois envisager plusieurs hypothèses afin de simplifier la syntaxe. L'énoncé initial indiquait que l'objet ne devait être construit que dans le tas. Cela entraîne par voie de conséquence, que l'objet ne peut être détruit que dans le tas. En modifiant l'énoncé du problème, on trouve une solution beaucoup plus élégante.

Construire un objet, uniquement dans le tas, permet d'éviter d'utiliser la pile avec des objets très volumineux.

Si vous désirez interdire également, toute dérivation de votre objet, déclarez-le destructeur en `private`.

Singleton

Il est parfois interdit d'avoir plusieurs instances d'une classe. Une classe n'ayant qu'une seule instance s'appelle un « singleton » [Gamma et al, DP:94]. Cet objet doit être global. Comment interdire la création de plusieurs instances globales ? Une première approche consiste à déclarer un attribut `static` à la classe, cet attribut étant justement une instance de celle-ci. Le constructeur étant protégé, seule la construction de cet attribut `static` peut être exécutée.

```
class CSingleton
{ protected:
  CSingleton();                              // Constructeur
  public:
  static CSingleton singleton;
```

```
  void f();
};

CSingleton CSingleton::singleton;           // Attribut static
```

L'utilisation de la seule instance possible de la classe `CSingleton` s'effectue comme cela :

```
CSingleton::singleton.f();
```

Le problème de cette approche est qu'il n'est pas possible de savoir si cette instance est construite avant son utilisation. Si un autre objet global utilise cette instance, cet autre objet peut être créé avant l'instance `CSingleton::singleton` (Voir la règle CPP.DEB.20, page 191). L'ordre d'initialisation de deux objets globaux dans deux fichiers différents n'est pas garantie.

Pour supprimer ce risque, il faut modifier légèrement la classe.

```
class CSingleton
{ protected:
  CSingleton();                             // Constructeur
  public:
  static CSingleton& singleton();
  void f();
};

CSingleton& CSingleton::singleton()
{ static CSingleton Ssingleton;
  return Ssingleton;
}
```

Par le mécanisme d'initialisation des variables statiques au sein d'une méthode, on est sûr que la variable est initialisée dès la première utilisation (Voir la règle CPP.STY.12, page 232). Le compilateur appellera le constructeur de `Ssingleton` lors du premier passage dans la méthode `singleton()`. L'appel du destructeur est également garantie par le compilateur. L'utilisation de cette instance est légèrement modifiée.

```
CSingleton::singleton().f();
```

Pour cacher l'existence de la méthode `singleton()` et faire croire à l'utilisateur qu'il s'agit d'une instance statique de classe, il faut utiliser le préprocesseur et modifier légèrement la classe.

```
class CSingleton
{ protected:
  CSingleton();                             // Constructeur
  public:
```

```
  static CSingleton& _singleton();
  void f();
};

CSingleton& CSingleton::_singleton()
{ static CSingleton Ssingleton;
  return Ssingleton;
}
#define singleton CSingleton::_singleton()
```

L'utilisation de cette instance devient alors classique.

```
singleton.f();
```

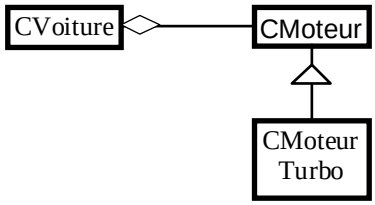
Si l'on désire instancier différentes dérivées de la classe `CSingleton` suivant une variable d'environnement, il faut écrire ce qui suit :

```
CSingleton& CSingleton::_singleton()
{ static CSingleton* pSingleton=NULL;
  if (pSingleton==NULL)
  { const char* pEnv=getenv("MODE");
    if (!strcmp(pEnv,"Trace"))
    { static CTrace Ssingleton;
      pSingleton=&Ssingleton;
    } else
    { static CNoTrace Ssingleton;
      pSingleton=&Ssingleton;
    }
  }
  return *pSingleton;
};
```

Seul le destructeur correspondant sera appelé lors de la fin du programme.

c. Constructeur de copie et objets polymorphes

Reprenons l'exemple du paragraphe précédent « Attribut virtuel » (Voir page 51), mais modélisons différemment les classes. Si l'on considère que l'objet `CVoiture` reçoit un `CMoteur`, la classe `CTurbo` n'est alors plus nécessaire. Une voiture possède des accessoires dont le moteur fait partie. Le constructeur de l'instance `CVoiture` reçoit un objet de type `CMoteur`. Il en devient le propriétaire, c'est-à-dire qu'il se chargera de la destruction de celui-ci. Il « adopte » un moteur.



La classe `CVoiture` devient ainsi :

```
class CVoiture
{ protected:
  CMoteur* _moteur;
  public:
    CVoiture(CMoteur* moteur=new CMoteur())
  : _moteur(moteur)
  { }
    ~CVoiture()
  { delete _moteur; }
};
```

Il est possible de construire une `CVoiture` de série ou de lui ajouter un moteur turbo.

```
CVoiture serie;
CVoiture turbo(new CMoteurTurbo());
```

L'objet `CVoiture` se charge de détruire l'objet moteur adopté par son destructeur. Si l'on décide de rédiger un constructeur de copie pour la `CVoiture`, il faut connaître le type de moteur à construire. Ce type n'est pas connu. La seule chose que connaît la `CVoiture` est qu'elle utilise un moteur qui lui est fourni. Comment construire une nouvelle instance du moteur du bon type pour dupliquer la `CVoiture` ? Faut-il appeler le constructeur de copie de `CMoteur` ou de `CMoteurTurbo` ?

Cette classe possédant un pointeur il faut rédiger un constructeur de copie (Voir la règle CPP.DEB.1, page 169). Une première approche consiste à déclarer celui-ci en `private`. Ainsi, pas de problème, il n'est pas possible de copier une `CVoiture`. Ce n'est pas une protection *légale* mais une impossibilité matérielle.

Une autre approche consiste à maintenir un attribut stipulant le type de classe à construire. Lors du constructeur de copie, le moteur correspondant au type est créé.

```
class CVoiture
{ protected:
  CMoteur* _moteur;
  public:
  enum TMoteur { Normal,Turbo } type;
    CVoiture(CMoteur* moteur=new CMoteur(),TMoteur t=CVoiture::Normal)
```

```
  : _moteur(moteur),
    type(t)
  { }
    CVoiture(const CVoiture& x)
  : type(x.type)
  { switch (x.type)
    { case Normal :
        _moteur=new CMoteur(*x._moteur);
        break;
      case Turbo :
        _moteur=new CMoteurTurbo(*(CMoteurTurbo*)x._moteur);
        break;
    }
  }
    ~CVoiture()
  { delete _moteur; }
};
```

Cette approche n'est absolument pas objet. Si vous désirez plus tard avoir un nouveau type de moteur, électrique par exemple, vous êtes obligé de modifier la classe `CVoiture` pour en tenir compte.

Dès que vous utilisez un type pour caractériser un pointeur sur une classe polymorphe, cherchez une solution utilisant les méthodes virtuelles.

Pour pouvoir résoudre ce cas, il serait intéressant d'avoir une sorte de constructeur de copie virtuel. Les constructeurs ne peuvent pas être virtuels. Pour dupliquer la classe, il faut le demander à l'objet `CMoteur`. Celui-ci connaît son propre type, il est capable de créer un `clone` de lui-même. Il faut ajouter une méthode virtuelle dans la classe `CMoteur` afin de pouvoir en construire un `clone`, et surcharger la fonction `clone()` (cf. « Clonage », page 42). Cette méthode sera redéfinie dans la classe `CMoteurTurbo` pour construire le `clone` correspondant.

```
class CMoteur
{ protected:
  virtual CMoteur* clone() const
  { return new CMoteur(*this); }
  public:
  virtual int nbCheveaux() const
  { return 10; }
  virtual ~CMoteur()
  { }
  friend inline CMoteur* clone(const CMoteur& x)
  { return x.clone(); }
};


class CMoteurTurbo : public CMoteur
{ protected:
```

```
  virtual CMoteur* clone() const[1]
  { return new CMoteurTurbo(*this); }
  public:
  virtual int nbCheveaux() const
  { return 25; }
};
```

Le constructeur de copie de `CVoiture` s'écrit alors comme ceci :

```
  CVoiture::CVoiture(const CVoiture& x)
: _moteur(clone(*x._moteur))
{ }
```

Cette approche est plus « propre » car elle permet l'ajout de nouveaux types de moteur sans modification de la classe `CVoiture`. Par contre la méthode `clone` doit exister dans la classe de base de l'attribut et être redéfinie dans toutes les classes dérivées. Le pointeur sur le moteur peut également être rédigé à l'aide d'un « pointeur d'agrégation » (Voir « Durée de vie des objets », page 57).

d. Smart pointer

Le C++ n'offre pas de mécanisme de gestion de mémoire permettant de récupérer la mémoire qui n'est plus utilisée. Chaque allocation doit explicitement être effacée. C'est un problème quotidien des développeurs, difficile à détecter, car le programme semble fonctionner correctement.

Il est possible, avec la puissance du C++, de construire un mécanisme de récupération automatique de la mémoire. Chaque objet alloué maintient un compteur de référence sur lui-même. Chaque fois qu'un pointeur pointe sur l'objet, ce compteur s'incrémente. Lorsqu'un pointeur ne pointe plus sur l'objet, le compteur est décrémenté. Lorsque ce compteur arrive à zéro, l'objet est automatiquement détruit. L'utilisateur n'a plus à gérer l'effacement des objets, cela se fait automatiquement.

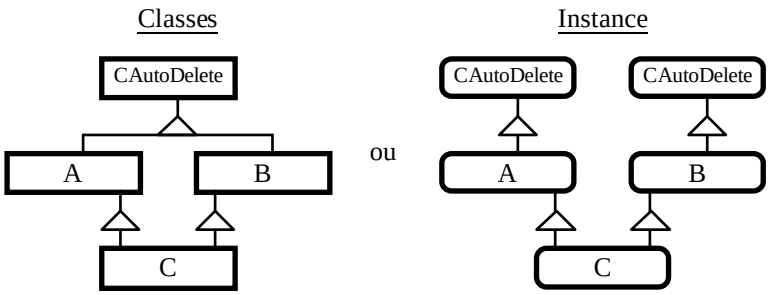
Pour utiliser cet outil il faut, dans un premier temps, que chaque objet offrant ce service possède un compteur de référence. La classe `CAutoDelete` s'occupe de cela. Tout objet voulant offrir ce service doit hériter de cet objet.

[1] ou `virtual CMoteurTurbo* clone() const` avec la nouvelle norme.

```
class CAutoDelete
{ public:
  unsigned _cntRef;
  protected:
    CAutoDelete()
  : _cntRef(0)
  { }
    ~CAutoDelete()
  { assert(!_cntRef); }
};
```

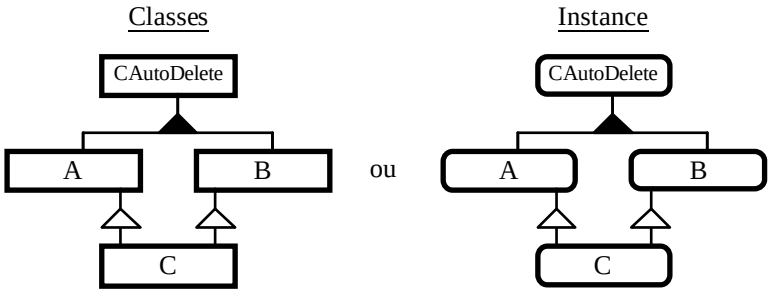
Attention, si une classe hérite de plusieurs objets offrant ce même service, la classe `CAutoDelete` sera présente deux fois dans l'objet.

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



Pour éviter cela, il faut que les classes `A` et `B` héritent virtuellement de `CAutoDelete`.

```
class A : virtual public CAutoDelete {};
class B : virtual public CAutoDelete {};
class C : public A, public B {};
```



Pour une dérivation simple, il n'est pas nécessaire d'hériter virtuellement.

L'objet possède maintenant un compteur de référence. Il faut gérer celui-ci automatiquement. Chaque fois qu'un pointeur pointe sur l'objet, il faut incrémenter la référence de l'objet. Pour cela, il faut contrôler les pointeurs sur l'objet. Nous allons créer un objet `template` simulant le comportement d'un pointeur. A chaque altération de celui-ci, les compteurs de référence des objets pointés seront ajustés.

```
// Smart Pointeur
template <class T>
class CSmartPt
{ T* _pt;
  void inc()
  { if (_pt!=NULL)
    { ++_pt->_cntRef;
    }
  }
  void dec()
  { if (_pt!=NULL)
    { if (!--_pt->_cntRef)
      { delete _pt;                          // Effacement de l'objet
        _pt=NULL;
      }
    }
  }
  public:
  // constructeur destructeur
    CSmartPt() : _pt(NULL) {}
    CSmartPt(T* x) : _pt(x)
  { inc(); }
    CSmartPt(const CSmartPt<T>& x) : _pt(x._pt)
  { inc(); }
    ~CSmartPt()
  { dec();
  }

  // Affectation
  CSmartPt<T>& operator =(const CSmartPt<T>& x)
  { if (&x!=this)
    { this->CSmartPt<T>::~CSmartPt();       // destructeur
      this->CSmartPt<T>::CSmartPt(x);       // constructeur de copie
      // ou new(this) CSmartPt<T>(x);
    }
    return *this;
  }
  CSmartPt<T> operator =(T* x)
  { this->CSmartPt<T>::~CSmartPt();         // dtr
    this->CSmartPt<T>::CSmartPt(x);         // cctr
    // ou new(this) CSmartPt<T>(x);
    return *this;
  }
```

```
  // Acces
  T* operator ->() const    { return _pt; }
  T& operator *() const     { return *operator ->(); }
  T& operator [](int index)
  { assert(!index); // Ne fonctionne pas avec un tableau
    return *operator ->();
  }
  operator T*() const       { return operator ->(); }
};
```

Pour accéder à l'objet, il faut utiliser le `CSmartPt<T>` à la place d'un pointeur. Ce `template` efface l'objet lorsqu'il n'existe plus de pointeur sur celui-ci.

```
class A : public CAutoDelete {};
typedef CSmartPt<A> PA;

void main()
{ PA p1;

  p1=new A();
  { PA p2=new A();
  } // Effacement de p2
  { PA p3=p1;
  } // N'efface pas p3
} // Effacement de p1
```

Ce *smart pointer* ne permet pas de pointer sur un tableau d'objet. En effet, le C++ ne fait pas la différence entre un pointeur sur un objet et un pointeur sur un tableau d'objets. C'est d'ailleurs source d'erreurs lors de l'utilisation des opérateurs `delete` (Voir « `::NEW` de debug », page 130).

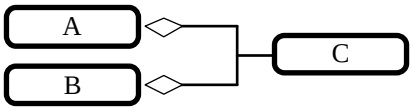
Pour interdire à un utilisateur d'obtenir une adresse sur un objet local ou global, afin de forcer l'utilisation de celui-ci avec un *smart pointer*, il faut ajouter à l'objet en `protected` un `operator &()` :

```
class A : public CAutoDelete
{ protected:
  A* operator &() { return this; }
  // ...
};
```

Avec celui-ci, il n'est pas possible d'initialiser un *smart pointer* avec l'adresse d'une variable locale.

```
void main()
{ A  a;
  PA p=&a; // Erreur
}
```

Ce *smart pointer* permet d'implanter une « agrégation partagée ». Si une instance possède un attribut qu'elle partage avec une autre instance, il faut résoudre la durée de vie de celui-ci. L'attribut doit être détruit lorsqu'il n'existe plus d'instance le possédant.



Si les instances A et B possèdent une relation d'agrégation vers la même instance C, les *smart pointer* permettent de gérer facilement la durée de vie de C.

```
class C : public CAutoDelete {};
typedef CSmartPt<C> PC;

class A
{ PC _c;
  //...
};

class B
{ PC _c;
  //...
};
```

e. Retour d'objet intermédiaire

Il est parfois nécessaire de construire un objet intermédiaire que l'utilisateur ne connaît pas, afin de cacher la complexité d'un service. Par exemple, si l'on désire faire un objet, utilisable comme un tableau, mais pourtant, qui manipule un fichier pour lire ou écrire des données.

Cet objet est dans un premier temps facile à concevoir.

```
class CFile
{ FILE* _stream;
  public:
    CFile(const char* filename)
  { _stream=::fopen(filename,"w+");
  }
    ~CFile()
  { ::fclose(_stream); }
  char operator [](int index) const
  { ::fseek(_stream,index,SEEK_SET);
    return ::fgetc(_stream);
  }
};
```

L'utilisateur de la classe accède aux caractères du fichier à l'aide de l'`operator []`. Il utilise ce fichier comme un tableau de caractères. L'`operator []() const` est facile à rédiger car l'utilisateur cherche à lire le fichier.

Si on désire offrir à l'utilisateur la possibilité d'écrire dans le fichier, cela devient beaucoup plus difficile. L'`operator []()` ne peut pas retourner une référence comme on le ferait si l'écriture était en mémoire. Il faut dans ce cas renvoyer un objet ayant redéfini l'`operator =()` pour détecter l'écriture dans le fichier. Cet objet doit également avoir un opérateur de conversion pour la lecture du fichier. Ce nouvel objet doit connaître l'objet `CFile` auquel il se rapporte, et la position courante dans le fichier où il faut écrire.

```
class CRetFile
{ CFile* _file;
  int    _index;
  public:
   CRetFile(CFile* file,int index)
  : _file(file), _index(index) {}
  CRetFile& operator =(char c)
  { ::fseek(_file->_stream,_index,SEEK_SET);
    ::fputc(c,_file->_stream);
    return *this;
  }
    operator char() const
  { ::fseek(_file->_stream,_index,SEEK_SET);
    return ::fgetc(_file->_stream);
  }
};
```

Cette classe n'ayant de raison d'exister qu'avec la classe `CFile`, elle doit être déclarée dans celle-ci. De plus, elle doit être `friend` de la classe `CFile` pour pouvoir accéder à l'attribut `_stream`.

Le listing complet est le suivant :

```
class CFile
{ FILE* _stream;

  class CRetFile
  { CFile* _file;
    size_t _index;
    public:
     CRetFile(CFile* file,size_t index)
    : _file(file), _index(index) {}
    CRetFile& operator =(char c)
    { ::fseek(_file->_stream,_index,SEEK_SET);
      ::fputc(c,_file->_stream);
      return *this;
    }


      operator char() const
```

```
    { ::fseek(_file->_stream,_index,SEEK_SET);
      return ::fgetc(_file->_stream);
    }
  };
  friend class CRetFile;
  public:
    CFile(const char* filename)
  { _stream=::fopen(filename,"w+");
  }
    ~CFile()
  { ::fclose(_stream); }
  char operator [](size_t index) const
  { ::fseek(_stream,index,SEEK_SET);
    return ::fgetc(_stream);
  }
  CRetFile operator[](size_t index)
  { return CRetFile(this,index);            // Retour obj intermediaire
  }
};
```

L'utilisateur de l'objet peut maintenant manipuler le fichier comme un tableau.

```
void main()
{ CFile f="tst.txt";

  f[0]='A';
  f[1]='B';
  f[2]=f[0];
  cout << f[0] << f[1] << endl;
}
```

La lecture et l'écriture sont effectuées à l'insu de l'utilisateur dans un fichier. Il voit cet objet comme une collection de caractères en mémoire.

H. LES ERREURS

a. Comment gérer les erreurs dans les constructeurs ?

Plusieurs approches sont possibles. Un constructeur ne renvoie pas de valeur. Il ne peut donc pas signaler qu'il a échoué.

État dans l'objet

Une première technique consiste à ajouter un état à l'objet indiquant si celui-ci est correctement construit.

```
class CObj
{ bool _status;
  public:
    CObj() : _status(false)
  { // ...
    _status=true;
  }
  bool isOk() const
  { return _status; }
};
```

Contexte de l'objet

Une autre approche consiste à utiliser les éléments déjà présents dans l'objet.

```
class CFile
{ FILE* _st;
  public:
    CFile(const char* name)
    : _st(NULL)
  { _st=fopen(name,"w"); }
    ~CFile()
  { fclose(_st); }
  int isOk() const
  { return (_st!=NULL);
  };
}
```

Le problème principal de cette approche, est que chaque méthode doit s'assurer que l'objet est correct avant d'en utiliser les éléments. Cela entraîne énormément de redondances à l'exécution.

```
void CFile::methode()
{ if (!isOk()) return;
  // ...
}
```

Il ne faut pas oublier qu'un objet peut être créé de façon temporaire par le compilateur pour y appeler une méthode ou un opérateur. La méthode doit s'assurer que l'objet est correct avant d'être exécutée. Il peut être prudent d'ajouter en tête de celle-ci une ligne `assert(isOk());` pour ne pas pénaliser le programme en mode exploitation.

Exceptions

La solution la plus pratique, est d'utiliser les exceptions. Une erreur lors d'un constructeur générera une exception (Voir « Exceptions », page 289).

```
class CObj
{ public:
    CObj()
  { // ...
    if (erreur) throw xError();
  }
};
```

Instance NULL

Pour détecter un objet en erreur, il est souvent utile d'avoir une instance particulière correspondant à une version de l'objet en *erreur*. Le C utilise cette technique pour indiquer qu'un pointeur est erroné. La valeur `NULL` indique qu'un pointeur n'est pas valide. Les allocateurs mémoire garantissent qu'aucune allocation ne sera faite à cette adresse particulière. `EOF` indique également un caractère invalide. Une fonction peut retourner cette valeur pour informer de son comportement anormal. Cela permet d'éviter de retourner deux variables : un caractère (ou un pointeur) et un code d'erreur. L'objet retourné, pointeur ou caractère, informe en lui-même de l'erreur possible.

Avec une classe, il peut être utile d'avoir ce même type de comportement. Une instance particulière peut informer de l'erreur d'une fonction. Par exemple, prenons une classe `CDate`. Nous désirons avoir une date particulière pour signaler une date erronée.

```
class CDate
{ int _jour,_mois,_annee;
  public:
    CDate(const char* str);
    CDate(int jour,int mois,int annee)
  : _jour(jour), _mois(mois), _annee(annee)
  { assert((jour>0) && (mois>0) && (annee>1900)); }
  int operator ==(const CDate& x) const
  { return ((_jour==x._jour) &&
            (_mois==x._mois) &&
            (_annee==x._annee));
  }
  static const CDate Null;
};
```

Nous désirons créer l'instance statique `CDate::Null` avec une valeur que ne peut pas fournir l'utilisateur de la classe. Pour cela, il faut ajouter un constructeur particulier en protégé, afin de pouvoir construire cette instance sans le test de postcondition (Voir « Tests », page 237).

```
class CDate
{ int _jour,_mois,_annee;
  protected:
  enum TError { error };
    CDate(TError)
```

```
  : _jour(0), _mois(0), _annee(0)
  { }
  public:
  static const CDate Null;
  //...
};
```

Il alors possible de construire l'instance `CDate::Null` à l'aide de ce constructeur. Le type de paramètre permet de sélectionner le constructeur désiré.

```
const CDate CDate::Null=CDate::error;
```

Maintenant, il est possible de retourner cette valeur lors d'une erreur d'une fonction.

```
CDate f()
{ //...
  if (erreur) return CDate::Null;
  return date;
}

void main()
{ if (f()==CDate::Null) cout << "Erreur dans f()" << endl;
}
```

Le test s'effectue sur cette valeur à l'aide de l'`operator ==()` de `CDate`. Étant donné que la seule instance pouvant avoir cette valeur est `NullDate`, il n'y a pas de confusion possible.

L'objet `CDate::Null` est constant car il ne faut pas laisser la possibilité à l'utilisateur de modifier cet objet. Il faut interdire une écriture comme

```
CDate::Null=CDate();
```

En déclarant cette instance constante, il n'est pas nécessaire d'ajouter une précondition dans les services pour interdire sa modification. Le compilateur est là à cet effet.

Il est parfois utile qu'il ne soit pas possible d'avoir plusieurs objets ayant simultanément la valeur `Null`. Il faut dans ce cas interdire de copier l'instance `Null` dans un autre objet.

```
class CDate
{ //...
    CDate(const CDate& x)
  { assert(&x!=&CDate::Null);
  }
  CDate& operator =(const CDate& x)
  { assert(&x!=&CDate::Null);
    //...
  }
  bool operator ==(const CDate& x) const
  { if ((&x==&CDate::Null) && (this==&CDate::Null)) return true;
    // ...
  }
};
```

Il faut ajouter des préconditions pour interdire la copie de `CDate::Null`. Il est alors possible d'optimiser l'opérateur de comparaison en vérifiant l'adresse particulière de `CDate::Null`.

Encapsulation dans une classe dérivée

S'il n'existe pas de combinaison de valeur pouvant être associée à la valeur `NULL`, il faut écrire une classe dérivée de `CDate`. Par exemple, si toutes les combinaisons de dates sont valides, l'objet `CDate` ne peut pas, en lui-même, indiquer une date invalide.

```
class CTstDate : public CDate
{ bool _erreur;
  public:
  enum TError { error };
  public:
    CTstDate(const CDate& x)
  : CDate(x), _erreur(false)
  { }
    CTstDate(TError)
  : CDate(1,1,1990), _erreur(true)
  { }
    operator bool() const
  { return _erreur; }
  bool operator !() const
  { return !_erreur; }
};
```

Une fonction ou une méthode désirant retourner une date ou une erreur, doit retourner un objet du type `CTstDate`.

```
CTstDate f()
{ //...
  if (erreur) return CTstDate(CTstDate::error);
  return CTstDate(date);
}
void main()
```

```
{ if (f()) cout << "Erreur dans f()" << endl;
}
```

C'est la même approche qu'utilise la fonction C `fgetc()` qui retourne un entier et non un caractère pour pouvoir retourner le code `EOF`. L'entier étant plus grand qu'un caractère, il peut contenir toutes les valeurs possibles d'un caractère. En quelque sorte, nous avons :

```
// Ce n'est pas du C++ !
class int : public char
{ ... };
```

Toutes les valeurs d'un caractère sont possibles. Il faut alors un type plus grand pour pouvoir contenir la valeur particulière `EOF`.

b. Traits de caractères

Pour pouvoir écrire un `template` avec une méthode retournant la valeur d'erreur d'un objet, il faut connaître son type. Une instance en erreur n'a pas forcement la même taille qu'une instance sans erreur. Par exemple, la valeur `EOF` qui indique un caractère incorrect, est stockée dans un entier et non dans un `char`. En effet, toutes les valeurs possibles d'un `char` sont valides. Il a fallu trouver une valeur supplémentaire à mettre dans un type de taille supérieure à `char`. La fonction `fgetc()` retourne un entier et non un `char` pour pouvoir vérifier la valeur `EOF`. Un `template` voulant retourner une valeur d'erreur, doit connaître les deux types possibles à manipuler : le type de base, et le type permettant de retourner une erreur. Imaginons que nous voulons écrire un `template` du type « pile » dont la méthode `pop()` retourne l'objet paramètre, ou une valeur d'erreur lorsque la pile est vide. Une pile de caractère retournera le caractère présent dans celle-ci ou `EOF` si la pile est vide.

```
template <class T>
class CStack
{ T _tab[10];
  int _nb;
  public:
    CStack() : _nb(0)
  { }
  void push(const T& data)
  { _tab[_nb++]=data;
  }
  <Type?> pop()
  { return (_nb) ? _tab[--_nb] : <Valeur?>;
};
```

Il faut connaître le type capable de recevoir la valeur d'erreur, et il faut connaître cette valeur d'erreur. Voici plusieurs combinaisons acceptables pour ce `template` :

Type de base	Type d'erreur	Valeur d'erreur
`char`	`int`	`EOF`
`wchart_t`	`wint_t`	`WEOF`
`int`	`int`	`INT_MAX`
`void*`	`void*`	`NULL`
`CDate`	`CDate`	`CDate::Null`
`CDate`	`CTstDate`	`CTstDate(error)`

Nous allons déclarer un `template` qui devra contenir les informations manquantes.

```
template <class T>
struct TError { };
```

Ce `template` est volontairement vide. Il permet simplement de signaler sa signature. Pour chaque type d'objet, il faudra rédiger une version spécifique de ce `template`.

```
struct TError<char>
{ typedef char type_base;
  typedef int  type_error;
  static inline type_error error()
  { return EOF; }
};

struct TError<int>
{ typedef int type_base;
  typedef int type_error;
  static inline type_error error()
  { return INT_MAX; }
};

struct TError<CDate>
{ typedef CDate type_base;
  typedef CTstDate type_error;
  static inline type_error error()
  { return type_error(error); }
};
```

Le `template` de la pile, utilisera `TError` pour générer le code correspondant au type utilisé.

```
template <class T>
class CStack
{ T _stack[10];
  int _sp;
  typedef TError<T>::type_error type_error;
```

```
  public:
    CStack() : _sp(0)
  { }
  void push(const T& data)
  { _stack[_sp++]=data;
  }
  type_error pop()
  { return (_nb) ? _stack[--_sp] : TError<T>::error();
  }
};
```

Une instance de `CStack<char>::pop()` retournera un entier. Une instance de `CStack<CDate>::pop()` retournera un objet `CTstDate`. Cette technique a été utilisée pour la première fois par le comité ANSI/ISO C++ pour rédiger les flux paramètrés.

c. Exception dans le constructeur de copie

Certaines situations sont difficiles à gérer. Une exception peut être appelée lors d'un constructeur de copie. Un constructeur de copie peut être appelé lors du retour d'une fonction. Il est difficile dans ce cas de maîtriser l'exception. Il n'est pas possible de la capturer.

```
class CObj
{ /*...*/
  public:
    CObj();
    CObj(const CObj& x);
    ~CObj();
};

class CStackObj
{ int   _sp;
  int   _max;
  CObj* _tab;
  // ...
  public:
    CStackObj();
    ~CStackObj();
  void push(const CObj& x);
  CObj pop();
};

CStackObj::CStackObj() : _sp(0), _max(0)
{ _tab=(CObj*)new char[sizeof(CObj)*10];
}

CStackObj::~CStackObj()
{ for (int i=_max-1;i>=0;--i)
  { _tab[i].CObj::~CObj();
  }
  delete [] (char*)_tab;
}
```

```
void CStackObj::push(const CObj& x)
{ assert(_sp<10);
  _tab[_sp++].CObj::CObj(x);
  // ou new (&_tab[sp++]) CObj(x);
  if (_sp>_max) _max=_sp;
  assert(_sp<10);
}

CObj CStackObj::pop()
{ return _tab[--_sp];
}
```

La méthode `CStackObj::pop()` retourne un objet extrait de la pile. Elle décrémente le pointeur de pile simultanément. Si une exception arrive dans le constructeur de copie de l'objet `CObj`, l'objet ne sera pas copié, mais le pointeur de la pile sera décrémenté. Le code de la méthode `pop()`est généré comme suit :

```
void CStackObj::pop(const CStackObj * const this,CObj* _ret)
{ --this->_sp;
  _ret->CObj::CObj(_tab[_sp]); // cctr, exception possible
}
```

Dans le cas d'une exception, `--this->sp` a été exécuté, ce que l'on cherche à éviter. Si le pointeur de la pile est décrémenté alors que la copie de l'élément a échouée, celui-ci est définitivement perdu. Comment trouver un moyen de modifier cet index, dans le cas où la copie a réussi ?

Il faut modifier `sp` après la copie. Dans la méthode, il n'est pas possible d'écrire un traitement exécuté après le `return`. Pour résoudre ce cas, il faut créer un objet supplémentaire permettant de modifier `sp` après la copie.

```
class CStackObjPop : public CObj
{ public:
    CStackObjPop(CStackObj& stack,CObj& obj)
  : CObj(obj)
  { --stack._sp; }
};
```

La méthode `pop()` retourne un objet de ce type qui est dérivé de `CObj`. Le constructeur de cet objet garde une relation avec l'objet `CStackObj`. C'est lors de l'appel du constructeur de copie de `CObj` que l'exception peut intervenir. Si l'objet `CStackObjPop` est correctement créé, l'utilisateur peut l'utiliser comme s'il s'agissait d'un objet `CObj`. Il peut écrire « `s.pop().a` » ou copier l'objet : « `CObj x=s.pop();` ». Si le constructeur de copie de `CObj` génère une exception, l'instruction « `--stack.sp` » ne sera pas exécutée.

Il faut modifier l'objet `CStackObj` comme suit :

```
class CStackObjPop;
class CStackObj
{ // ...
  public:
  friend class CStackObjPop;
  CStackObjPop pop();
};

CStackObjPop CStackObj::pop()
{ return CStackObjPop(*this,_tab[_sp-1]);
}
```

d. Exception dans un template

Un `template` utilise des méthodes de l'objet générique. Celles-ci peuvent être amenées à générer une exception. Il faut, lors de la rédaction d'un `template`, identifier tous les appels des méthodes de l'objet générique et se poser la question du comportement du `template` si une exception arrive. Cela comprend, les constructeurs, les opérateurs, les conversions, et surtout, le constructeur de copie. Il faut penser à tout ce que le compilateur fait derrière « notre dos » pour identifier correctement l'appel de toutes les méthodes de l'objet générique.

```
template <class T>
class CList
{ struct CNode
  { CNode* _next;
    T      _obj;
      CNode(CNode* next,const T& obj)
    : _next(next), _obj(obj) {}             // cctr de T
  };
  CNode* _first;
  public:
    CList() : _first(NULL) {}
  CList<T>& operator +=(const T& x)
  { _first=new CNode(_first,x);
    return *this;
  }
  CList<T>& operator -=(const T& x)
  { CNode* opt=NULL;
    for (CNode* pt=_first;pt!=NULL;opt=pt,pt=pt->_next)
    { if (pt->_obj==x)                      // operator == de T
      { (opt==NULL) ? _first : opt->_next=pt->_next;
        delete pt;                          // dtr de T
      }
    }
    return *this;
  }
};
```

Chaque utilisation d'une méthode de `T` peut générer une exception. Cette classe doit être entièrement réécrite pour en tenir compte.

I. CLASSE MUTANTE

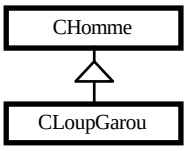
Il est parfois nécessaire d'avoir un objet mutant. Celui-ci est d'un type particulier à un moment de son existence, puis il se transforme en un autre type par la suite. Cet objet se transforme en un autre objet.

Pour prendre un exemple concret, imaginons la modélisation d'un « loup-garou ». Dans les contes fantastiques, un loup-garou est un *homme* qui se transforme en un *loup* les nuits de pleine lune. Plusieurs approches sont possibles pour modéliser cette transformation. L'objectif est de transformer une instance de type `Homme` en une instance de type `Loup`. La première approche consiste à détruire l'instance `Homme` et de construire une nouvelle instance `Loup` à la place. Cette approche est difficile à mettre en place. En effet, les objets sont identifiés par leurs adresses. La nouvelle instance `Loup` n'est pas à la même adresse que l'instance `Homme` précédente. Pour le système, cela ne correspond pas à une mutation de l'objet, mais à la création d'un nouvel objet et la destruction du précédent. Toutes les références sur l'objet `Homme` deviennent invalides après la mutation. Pour écrire correctement cette approche, il faut maintenir l'ensemble des relations avec l'objet `Homme` et les mettre à jour après la mutation. C'est très lourd et compliqué. Le risque de pointeurs flottants est alors très important.

a. Mutation avec héritage

D'autres approches permettent de modéliser réellement la mutation. Tout dépend des caractéristiques de mutation. Si l'objet mutant peut être décrit par la classe héritée, la meilleure approche est de ventiler les méthodes virtuelles suivant l'état courant de la mutation.

```
class CHomme
{ public:
  virtual ~CHomme()
  {}
  virtual void alimentation() const
  { cout << "omnivore" << endl; }
  virtual void profession() const
  { cout << "journaliste" << endl; }
};

class CLoupGarou : public CHomme
{ enum {THomme,TLoup} _mode;

  public:
  CLoupGarou() : _mode(THomme) {}
```



```
  virtual void alimentation() const
  { if (_mode==TLoup) cout << "carnivore" << endl;
    else CHomme::alimentation();
  }
  void profession() const
  { assert(_mode==THomme);
    CHomme::profession();
  }
  void mutation()
  { _mode=(_mode==THomme) ? TLoup : THomme;
  }
};

void main()
{ CLoupGarou paul;

  paul.alimentation();
  paul.profession();
  paul.mutation();
  paul.alimentation();
}
```

La classe `CLoupGarou` maintient un mode lui permettant de modifier le comportement des méthodes virtuelles. Ces méthodes appellent les versions héritées, ou bien, elles en modifient le comportement. Dans l'exemple précédent, la méthode `alimentation()` est résolue à l'exécution suivant l'état courant de l'instance. Une méthode particulière de l'objet permet de changer l'état de l'instance, de muter l'objet.

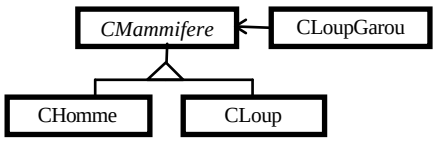
La méthode `profession()` ne peut être appelée que si l'instance mutante est du type `CHomme`. Un `assert()` vérifie cette précondition. La demande de `profession()` d'un loup-garou les nuits de pleine lune entraîne l'arrêt du système pour incohérence.

Si la mutation doit s'effectuer entre deux classes, l'approche précédente n'est pas valide. La classe `CLoup` n'est pas présente dans cet exemple. La classe `CHomme` n'est pas transformée en une classe `CLoup`. C'est le mode de la classe `CLoupGarou` qui identifie la mutation.

b. Mutation avec association

Pour résoudre une mutation entre deux classes, une des approches consiste à maintenir une relation avec l'instance réelle de l'objet. Pour la mutation de la classe `CHomme` en classe `CLoup`, la classe mutante offre une interface sur ces deux classes.

```
class CMammifere
{ public:
  virtual void alimentation() const =0;
  virtual ~CMammifere()
  {}
};
```



```
class CHomme : public CMammifere
{ public:
  virtual void alimentation() const
  { cout << "omnivore" << endl; }

  virtual void profession() const
  { cout << "journaliste" << endl; }
};

class CLoup : public CMammifere
{ public:
  virtual void alimentation() const
  { cout << "carnivore" << endl;
  }
};

class CLoupGarou
{ enum {THomme,TLoup} _mode;
  CMammifere*         _instance;

  public:
    CLoupGarou() : _mode(THomme), _instance(new CHomme()) {}
  void alimentation() const
  { _instance->alimentation();
  }
    ~CLoupGarou()
  { delete _instance;
  }
  void profession() const
  { assert(_mode==THomme);
    ((CHomme*)_instance)->profession();
  }
  void mutation()
  { delete _instance;
    if (_mode==THomme)
    { _mode=TLoup;
      _instance=new CLoup();
    }
    else
    { _mode=THomme;
      _instance=new CHomme();
    }
  }
};

void main()
{ CLoupGarou paul;

  paul.alimentation();
  paul.profession();
  paul.mutation();
  paul.alimentation();
}
```

Un pointeur sur la classe de base est présent dans la classe mutante. C'est l'adresse de cette instance mutante qui identifie l'objet. Cette classe manipule deux types d'instances suivant l'état courant de l'objet. La mutation se traduit par la destruction de l'instance `CHomme` et par la création d'une instance `CLoup`. L'ensemble des méthodes offertes par les deux objets `CHomme` et `CLoup` est redéfini pour sous-traiter celles-ci à l'instance associée. La classe `CLoupGarou` est un « proxy » des deux classes.

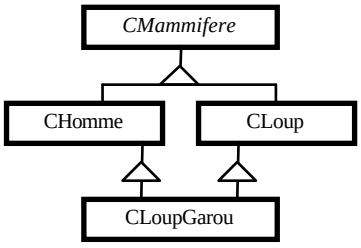
L'inconvénient de cette approche est qu'il n'est pas possible d'utiliser le polymorphisme entre les classes `CLoupGarou`, `CLoup` et `CHomme`. Ces classes ne sont pas en relation d'héritage. Les méthodes de `CLoupGarou` *simulent* les méthodes de `CLoup` et de `CHomme`. Un `CLoupGarou` n'est pas une sorte de `CLoup` ni une sorte d'`CHomme`.

c. Mutation avec héritage multiple

Pour régler la mutation en gardant les relations d'héritages, il faut utiliser une nouvelle approche qui permet de garder les capacités de polymorphisme entre les classes.

La classe `CLoupGarou` doit hériter des deux classes `CHomme` et `CLoup`. Un `CLoupGarou` est une sorte d'homme et une sorte de loup, mais pas en même temps. Nous cherchons à implanter une sorte d'héritage dynamique d'instance. Chaque instance désire indiquer la classe qu'elle hérite à un moment donné de son existence.

```
class CMammifere
{ public:
  virtual void alimentation() const =0;
  virtual ~CMammifere()
  {}
};

class CHomme : public CMammifere
{
  public:
  virtual void alimentation() const
  { cout << "omnivore" << endl; }
  virtual void profession() const
  { cout << "journaliste" << endl; }
};

class CLoup : public CMammifere
{ public:
  virtual void alimentation() const
  { cout << "carnivore" << endl;
  }
};

class CLoupGarou : public CHomme, public CLoup
{ public:
  enum {THomme,TLoup} _mode;
```



```
  CLoupGarou() : _mode(THomme) {}
  void alimentation() const
  { (_mode==THomme) ? CHomme::alimentation() : CLoup::alimentation();
  }
  void profession() const
  { assert(_mode==THomme);
    CHomme::profession();
  }
  void mutation()
  { _mode=(_mode==THomme) ? TLoup : THomme;
  }

};

void main()
{ CLoupGarou paul;
  CHomme* pHomme;
  CLoup* pLoup;

  pHomme=&paul;
  paul.alimentation();
  paul.profession();

  paul.mutation();
  paul.alimentation();

  pLoup=&paul;
  pHomme=&paul;                          // conversion incorrecte
  pHomme->alimentation();
  pHomme->profession();                  // conversion incorrecte
}
```

La classe `CMammifere` peut être héritée virtuellement par `CHomme` et `CLoup` si les caractéristiques `CMammifere` du loup-garou ne se modifient pas lors de la mutation. Cette écriture permet de convertir un `CLoupGarou` en `CHomme` ou en `CLoup`. Le polymorphisme est respecté. Par contre, la conversion n'est valide que lors d'un état particulier de l'instance. Il ne devrait pas être possible de convertir un `CLoupGarou` en `CHomme` lors des nuits de pleine lune. Il serait intéressant d'offrir une protection à l'exécution du type :

```
CHomme* operator &()
{ assert(_mode==THomme);
  return this;
}
CLoup* operator &()
{ assert(_mode==TLoup);
  return this;
}
```

mais cela ne fonctionne pas car le langage ne peut pas différencier deux méthodes par leurs codes retour. Il faut alors ajouter un intermédiaire s'occupant de vérifier la validité de la conversion lors de l'exécution. L'`operator &()` doit retourner un objet ayant deux

opérateurs de conversion vers les deux types possibles de la mutation, `CHomme` et `CLoup`. Cet objet intermédiaire vérifie la validité de la conversion.

Nous allons ajouter une classe `PLoupGarou` dans la classe `CLoupGarou`. Celle-ci doit être `private` car personne d'autre que `CLoupGarou` ne doit pouvoir en créer une instance. Cette classe possède un pointeur vers un `CLoupGarou`. L'`operator &()` va alors retourner une instance de `CPLoupGarou`.

```
class CLoupGarou : public CHomme, public CLoup
{ //...
  private:
  class CPLoupGarou
  { CLoupGarou* _pLoupGarou;
    public:
      CPLoupGarou(CLoupGarou* pLoupGarou)
    : _pLoupGarou(pLoupGarou) {}
      operator CHomme* ()
    { assert(_pLoupGarou->_mode==THomme);
      return _pLoupGarou;
    }
      operator CLoup* ()
    { assert(_pLoupGarou->_mode==TLoup);
      return _pLoupGarou;
    }
  };
  public:
  CPLoupGarou operator &()
  { return CPLoupGarou(this); }
};
```

La récupération de l'adresse d'une instance de `CLoupGarou` est vérifiée à l'exécution. Le mode courant de l'instance doit être valide. Par exemple :

```
{ CLoupGarou LoupGarou;
  CLoup* p=&LoupGarou; // Erreur
}
```

Malheureusement, cette technique ne fonctionne pas dans tous les cas. Un pointeur `CLoupGarou*` peut être converti en `CLoup*` sans vérification.

```
CLoupGarou* pLoupGarou=&paul
CLoup*      pLoup=pLoupGarou; // Valide même si mode=THomme
```

Pour utiliser l'opérateur déclaré précédemment, il faut utiliser une écriture inhabituelle

```
CLoup*     pLoup=&*pLoupGarou; // Verification de la conversion
```

Si les méthodes vérifient le contexte courant de l'instance en précondition, les conversions erronées seront détectées à l'exécution des méthodes mais pas lors de la conversion. Il n'est alors pas nécessaire d'ajouter l'artifice précédent. Celui-ci étant incomplet comme on vient de le voir.

Cette technique doit rester rarissime. Il est préférable de modifier la conception que de devoir écrire ce type de modélisation. La plupart des mutations peuvent se régler à l'aide de l'approche modale expliquée au paragraphe « Mutation avec héritage ».

J. ÉCRITURES GENERIQUES

Voici des écritures permettant de garantir une cohérence sémantique.

```
inline void* operator new(size_t,void* p)   { return p; }

class CObj
{
// Constructeur de copie et assignation
  CObj(const CObj& x)                       { /*...*/ }
  CObj& operator =(const CObj& x)
  { if (&x!=this)
    { this->CObj::~CObj();  // Efface this
      ::new (this) CObj(x); // cctor
    }
    return *this;
  }


// Test booleen unaire
  operator const void*() const[1]           { /*...*/ }
  int operator !() const                    { return !*this; }
  ou
  operator bool() const                 { /*...*/ }
  bool operator !() const               { return !*this; }

// Test booleen binaire
  bool operator ==(const CObj& x) const     { /*...*/ }
  bool operator !=(const CObj& x) const     { return !(*this==x); }
  bool operator <(const CObj& x) const      { /*...*/ }
  bool operator >=(const CObj& x) const     { return !(*this<x); }
  bool operator >(const CObj& x) const      { return (x<*this); }
  bool operator <=(const CObj& x) const     { return !(*this>x); }

// Smart pointer
  CObj* operator ->() const                 { /*...*/ }
  CObj& operator *() const                  { return *operator->(); }
```

[1] ou `operator bool () const` avec la future norme.

```
  CObj& operator [](int i) const            { return operator->()[i]; }

// Operation
  CObj& operator +=(const CObj& z)          { /*...*/ }
  CObj operator +(const CObj& z) const      { return CObj(*this)+=z }
  CObj& operator ++()                       { /*...*/ }
  CObj operator ++(int)
  { CObj tmp=*this;
    ++*this;
    return tmp;
  }
  friend CObj operator +(int i,const CObj& x) // Si associat.
  { return x+i; }

  // Clonage
  private:
  CObj* clone() const
  { return new CObj(*this); }
  friend CObj* clone(const CObj& x)
  { return x.clone(); }
};
```

Pour chaque méthode indiquée avec « `/*...*/` », d'autres s'appuient sur elle, pour leur implantation. Il est fortement recommandé de s'inspirer de ces écritures afin de ne pas dupliquer le code, et de garder une cohérence sémantique entre les méthodes.

Attention, toute classe dérivée doit redéfinir l'`operator =()` comme indiqué ci-dessus. Si votre compilateur ne possède pas encore le type `bool` remplacez celui-ci par `int`.

Les « Standard Template Library » de [Stepanov et al, STL:95] introduisent des opérateurs `template` traduisant les sémantiques équivalentes des opérateurs booléens.

```
template <class T>
inline bool operator !=(const T& x,const T& y)
{ return !(x==y); }
template <class T>
inline bool operator >(const T& x,const T& y)
{ return y < x; }
template <class T>
inline bool operator <=(const T& x,const T& y)
{ return !(y < x); }
template <class T>
inline bool operator >=(const T& x,const T& y)
{ return !(x < y); }
```

Avec ces opérateurs, il suffit de rédiger les `operator ==()` et `operator <()` pour bénéficier directement des autres. Un comportement spécifique peut toujours être ajouté pour ces opérateurs supplémentaires, mais par défaut ils existent. D'autres opérateurs du même type peuvent être rédigés :

```
template <class T>
inline T operator +(const T& x,const T& y)
{ return T(x)+=y; }
template <class T>
inline T operator -(const T& x,const T& y)
{ return T(x)-=y; }
...
```

CHAPITRE 4

AUTRES PRINCIPES DE CODAGE

Vous trouverez dans ce chapitre différents utilitaires d'aide au développement, ainsi que certains points de vue particuliers sur l'utilisation du C++.

A. COMMENT OPTIMISER LA COMPILATION ?

- Le compilateur doit, pour chaque `include`, interpréter la syntaxe et maintenir un ensemble d'informations pour pouvoir compiler le programme. Si vous déclarez une classe qui n'utilise dans le « `.h` » qu'un pointeur ou une référence sur un objet, il n'est pas nécessaire d'inclure le fichier correspondant à cet objet. Une simple déclaration du nom de l'objet est suffisante pour compiler. Dans ce cas, le compilateur n'a pas à maintenir l'ensemble des informations sur cet objet. Seule l'existence de celui-ci lui importe.

```
class Use;
class A
{ Use* _p;
  Use& _r;
  public:
    A(Use* p,Use& r) : _p(p), _r(r) {}
};
```

- Le compilateur doit également ouvrir les fichiers `include` à chaque demande. Si vous ajoutez un mécanisme pour empêcher les inclusions multiples des fichiers `include`, vous pouvez encadrer les `include` de vos applications par un `#ifndef`, afin d'éviter au compilateur d'ouvrir un fichier, puis de constater qu'il n'a rien à en faire, avant de le fermer.

Fichier lib.h :

```
#ifndef LIB_H
#define LIB_H
// ...
#endif
```

Fichier main.c :

```
#ifndef LIB_H
#include "lib.h"
#endif
// ...
```

- Une méthode `private` et `inline` n'a pas besoin d'être présente dans le fichier en tête si elle n'est pas utilisée par une autre méthode `inline`. Il faut déclarer cette méthode dans le fichier source. Comme cela, les utilisateurs du fichier en tête ne demanderont pas au compilateur d'interpréter cette méthode qui ne peut pas être appelée, et d'autre part, la modification du code de celle-ci n'entraînera pas la recompilation des fichiers utilisant cette classe. Seule l'édition de liens sera impactée.

a. Modifier une classe sans tout recompiler

Le C++ utilise la taille de chaque objet pour l'allocation. Pour cela, il faut que le compilateur connaisse entièrement la classe. Il doit connaître les méthodes appartenant à une classe et les données `private` ou `public` pour pouvoir réserver la place nécessaire à l'utilisation de celle-ci.

Une modification mineure de la classe peut entraîner une cascade de recompilation, car cette modification peut avoir des conséquences sur un ensemble de classes. Il serait souhaitable de séparer les méthodes des attributs de la classe.

Une technique permet d'éviter les recompilations. Pour chaque classe souvent modifiée, il faut déclarer une classe d'interface (ou proxy) et une classe de corps. La classe d'interface manipule l'objet *via* un pointeur. Les modifications n'étant faites que sur le corps de la classe, le compilateur ne doit pas alors tout recompiler.

La classe `CVehicule` suivante :

```
class CVehicule
{ char* _nom;
  int   _nbRoues;
  public:
    CVehicule(const char* nom,int nbRoues)
  : _nbRoues(nbRoues)
  { _nom=new char[strlen(nom)+1];
    strcpy(_nom,nom);
  }
    ~CVehicule()
  { delete [] _nom; }
  const char* nom() const
  { return _nom; }
  int nbRoues() const
  { return _nbRoues; }
};
```

peut être modifiée comme suit :

Fichier `vehicule.h` :

```
class CVehiculeCorps;
class CVehicule
{ CVehiculeCorps* _corps;
  public:
      CVehicule(const char* nom,int nbRoues);
      ~CVehicule();
    const char* nom() const;
    int nbRoues() const;
};
```

Fichier `vehicule.cxx` :

```
class CVehiculeCorps
{ char* _nom;
  int   _nbRoues;
  public:
    CVehiculeCorps(const char* nom,int nbRoues)
  : _nbRoues(nbRoues)
  { _nom=new char[strlen(nom)+1];
    strcpy(_nom,nom);
  }
    ~CVehiculeCorps()
  { delete [] _nom; }
  const char* nom() const
```

```
  { return _nom; }
  int nbRoues() const
  { return _nbRoues; }
};

// Implantation de l'interface
CVehicule::CVehicule(const char* nom,int nbRoues)
{ _corps=new CVehiculeCorps(nom,nbRoues); }
CVehicule::~CVehicule()            { delete _corps; }
const char* CVehicule::nom() const { return _corps->nom(); }
int CVehicule::nbRoues() const     { return _corps->nbRoues(); }
```

La nouvelle version de `CVehicule` implante l'interface avec la classe `CVehiculeCorps`. Lors de la modification de `CVehiculeCorps`, seul le fichier `vehicule.cxx` doit être recompilé. L'ajout d'un attribut n'entraîne pas la recompilation de tout le programme. Les langages Eiffel, Smalltalk ou Objective C, utilisent cette technique, certains compilateurs C++ également. Avec cette approche, seules les méthodes de création des objets doivent être recompilées. Les clients de la classe ne font que manipuler des pointeurs, sans le savoir. Il n'est plus nécessaire de recompiler toutes les librairies utilisant une classe lors de la modification de celles-ci.

Par contre, il n'est plus possible de bénéficier des méthodes `inline`. Chaque modification de l'interface de la classe de corps doit être indiquée dans la classe d'interface.

Par la suite, pour des raisons de performance, lorsque la classe devient stabilisée, vous pouvez supprimer la classe d'interface, et utiliser directement la classe de corps. Cette astuce est à utiliser en cours de développement, mais à un coût non nul lors de l'exécution. Un mécanisme de `#define` peut implanter les deux versions suivant la phase de développement en cours.

b. Compiler les `template`

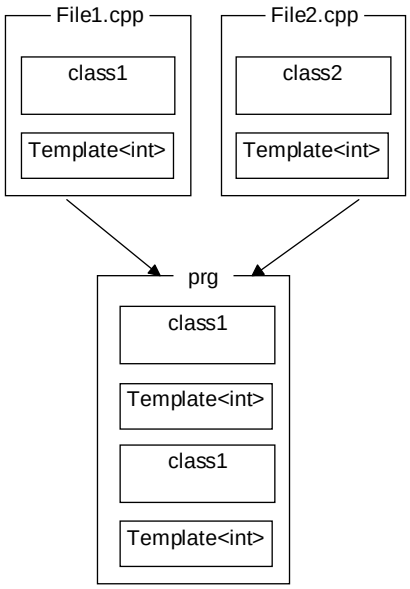
Les `template` offrent la possibilité de concevoir des classes ou des fonctions indépendantes du type des objets manipulés. Typiquement, un objet *conteneur* doit maintenir une collection d'objets. Les algorithmes permettant de gérer cette collection sont indépendants de l'objet manipulé. Malheureusement, avec un langage fortement typé comme le C++, il n'est pas possible directement de décrire une classe *conteneur* indépendante du type. Il faut rédiger une version pour chaque type d'objet. Toutes ces versions étant similaires à l'exception du type utilisé. Les `template` permettent de décrire cette classe à l'aide d'un type générique. Le compilateur générera automatiquement les versions nécessaires suivant les types rencontrés. Auparavant, les développeurs utilisaient pour cela le préprocesseur. Les classes étaient alors très difficiles à corriger ou à modifier.

Une particularité des `template` est de devoir être instanciés. Instancier un `template` indique qu'il faut générer une version particulière de celui-ci pour un type donné d'objet.

L'instanciation d'un `template` dans un fichier déclare dans le fichier objet les méthodes nécessaires à l'utilisation de celui-ci. Si un autre fichier décide d'instancier le même `template` et que les deux fichiers sont liés entre eux, le `link` se retrouve avec plusieurs versions des mêmes méthodes.

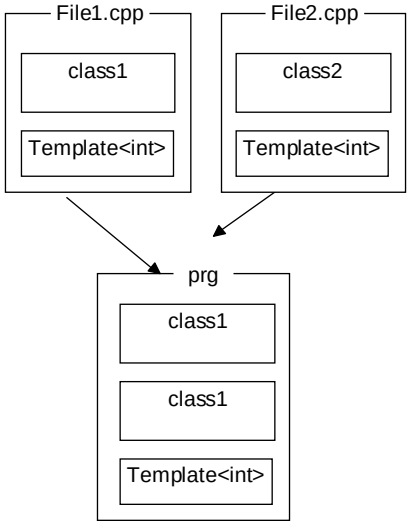
Ce problème est identique à l'instanciation des tables de sauts virtuels. Chaque utilisation d'une classe ayant des méthodes virtuelles génère une table de sauts. Cette table peut être présente dans plusieurs fichiers. Lors du `link`, il ne doit y avoir qu'une seule table.

Pour résoudre cette instanciation, les compilateurs utilisent plusieurs stratégies. Premièrement, l'instanciation du `template` est effectuée lors de chaque utilisation de celui-ci. Chaque fichier utilisant un `template` l'instancie. Pour éviter la présence de plusieurs versions des mêmes méthodes dans plusieurs fichiers, celles-ci sont déclarées en statique. L'instanciation du `template` est faite en statique dans chaque fichier. La liaison de plusieurs fichiers utilisant le même `template` ne pose plus de problème au `link`. Par contre, cela alourdit considérablement le programme. Celui-ci possède plusieurs fois les mêmes méthodes qui auraient pu être partagées.



Une autre approche consiste à déclarer les `template` avec un attribut particulier dans le fichier objet. Le Fortran permet d'avoir un tableau en *common*. Cela veut dire que plusieurs fichiers différents peuvent déclarer différemment le même tableau. Un fichier peut par exemple déclarer un tableau `T` avec deux éléments, tandis qu'un autre fichier déclare le

même tableau avec dix éléments. Lors de la liaison de ces deux fichiers, le `link` détecte ces deux versions du même tableau et sélectionne la version de plus grande taille. Ces tableaux utilisent un champ particulier appelé *common*, géré spécifiquement par le `link`. Certains compilateurs C++ génèrent les `template` avec cet attribut. Le `link` détecte plusieurs versions des mêmes modules de même taille (normalement), et en sélectionne un pour lier le programme. Le même `template` peut être instancié dans plusieurs fichiers, une seule version sera présente dans le programme final.



Les préprocesseurs C++ ne peuvent pas utiliser cet artifice. En effet, il n'existe pas de mécanisme de *common* en C ANSI. L'approche statique est alors préférable. Bjarne Stroustrup propose une approche différente [Stroustrup, DE:94]. Les `template` ne sont jamais instanciés. Lors du `link`, les `template` non résolus apparaissent. Le compilateur décide alors de reprendre les sources des `template` et de lancer la compilation pour instancier ceux-ci. L'édition de lien est alors adaptée pour y ajouter les nouveaux fichiers objets. Ce mécanisme recommence jusqu'à ce qu'il n'y ait plus de `template` non instanciés.

Comme on vient de le voir, il faut pouvoir n'avoir qu'une seule version de l'instanciation d'un `template`. Une autre approche consiste à laisser au développeur la gestion de l'instanciation du `template`. Le développeur a la charge de faire en sorte qu'une seule version du `template` soit présente dans le programme. Pour cela, une option du compilateur permet de demander l'instanciation des `template` rencontrés. Dans le cas contraire, les `template` ne sont pas instanciés. Un `#pragma` peut également jouer ce

rôle. L'utilisateur doit adapter les options de compilation des différents modules de son programme pour ne générer qu'une seule version de l'instanciation d'un `template`.

Certains compilateurs déclarent dans un fichier particulier la liste des `template` qu'ils rencontrent. Ce fichier s'incrémente au fur et à mesure de la compilation de chaque module. Si l'instanciation d'un `template` est déjà déclarée dans ce fichier, celle-ci ne sera pas ajoutée. Cette approche a un inconvénient. Au fur et à mesure de votre développement, ce fichier va accumuler tous les `template` nécessaires. Si vous déclarez un `template` et que vous le modifiez par la suite, l'ancienne version sera toujours présente dans ce fichier. Il faut régulièrement l'effacer et recompiler tous les fichiers de votre programme pour ne garder que les `template` réellement utiles à votre application.

La future norme du C++ déclare une nouvelle syntaxe permettant de demander l'instanciation d'un `template` particulier.

```
template class List<int>;                  // Classe
template int List<int>::operator [](int);  // Méthode
template int max<int,int>(int,int);        // Fonction
```

Pour pouvoir instancier un `template` le compilateur a besoin du source complet de celui-ci. En développement classique, vous rédigez un fichier « en tête » avec la déclaration de vos classes et les méthodes `inline` associées. Dans un autre fichier vous déclarez les méthodes non `inline`. Pour l'instanciation d'un `template`, ces deux fichiers sont nécessaires. Il ne serait pas raisonnable de déclarer toutes les méthodes d'un `template` en `inline`. La compilation serait dans ce cas très longue. Il faut séparer les déclarations `inline` d'un `template` des déclarations non `inline`, comme vous le faites pour des classes standard. Par contre, il faut offrir la possibilité de réunir ponctuellement ces deux fichiers lors de l'instanciation d'une version du `template`. Une seule instanciation de celui-ci dans l'application a besoin de la déclaration complète du `template`. Vous devez gérer comme indiqué précédemment cette instanciation. Un `#define` peut vous permettre d'inclure ou non dans le fichier « en tête » du `template` le fichier source de celui-ci. Pour éclaircir cela, prenons un exemple.

```
template <class T,int max>
class CBoundArray
{ T _tab[max];
  public:
    CBoundArray() {}
  T& operator [](int index)
  { assert(index<max);
    return _tab[index];
  }
  int find(const T& x) const;
};

template <class T,int max>
int CBoundArray<T,max>::find(const T& x) const
```

```
{ int i=0;
  for (;i<max;++i)
    if (_tab[i]==x) return i;
  return -1;
}
```

Celui-ci permet de déclarer un tableau dont l'indexation est vérifiée. L'utilisateur ne peut pas utiliser un index plus grand que la taille du tableau. Un service supplémentaire est ajouté pour retrouver l'index d'une valeur. La méthode `find` du `template` n'est pas une méthode `inline`. Pour instancier ce `template`, le compilateur a besoin de connaître entièrement celui-ci, la méthode `find` également. Pourtant, seules les méthodes `inline` sont nécessaires à la compilation de la classe lors de son utilisation. Il faut alors organiser cela pour pouvoir choisir d'inclure ou non, entièrement le `template`. Il faut le déclarer dans deux fichiers. Le premier ne possédant que la déclaration de la classe `template` et le second possédant la déclaration de la méthode `find`.

Fichier BndArray.h :

```
template <class T,int max>
class CBoundArray
{ T _tab[max];
  public:
    CBoundArray() {}
  T& operator [](int index)
  { assert(index<max);
    return _tab[index];
  }
  int find(const T& x) const;
};

#ifdef COMPILE_INSTANTIATE
#include "BndArray.cxx"
#endif
```

Fichier BndArray.cxx :

```
template <class T,int max>
int CBoundArray<T,max>::find(const T& x) const
{ int i=0;
  for (;i<max;++i)
    if (_tab[i]==x) return i;
  return -1;
}
```

Pour instancier entièrement ce `template` une fois dans le programme, il faut compiler avec le `#define COMPILE_INSTANTIATE`. Vous devez gérer les instanciations des `template` nécessaires à votre application, et n'ajouter ce `#define` qu'une seule fois dans le programme. Cela va considérablement accélérer les compilations de votre programme. Seules les informations strictement nécessaires seront présentes dans le fichier

« en tête ». Sans cette approche, les `template` ont l'inconvénient d'être très gourmands en temps de compilation.

c. Quand recompiler ?

Il n'est pas nécessaire de recompiler une classe si :

- Vous ajoutez une méthode non `virtual`.
- Vous modifiez une méthode non `inline`.
- Vous déclarez une classe dérivée d'une classe de base.
- Vous ajoutez un constructeur.
- Vous ajoutez une méthode ou un attribut `static`.

Il faut recompiler toute la hiérarchie si :

- Vous modifiez une méthode `inline`
- Vous ajoutez ou supprimez un attribut d'un objet.
- Vous ajoutez une méthode `virtual`.
- Vous modifiez l'héritage.
- Vous ajoutez un constructeur par défaut.
- Vous ajoutez un opérateur d'affectation.
- Vous ajoutez un destructeur.

B. QUAND ET OU UTILISER LES REFERENCES ?

Les anciens développeurs C préfèrent les pointeurs aux références car ils les maîtrisent déjà. Les références ont été ajoutées au langage pour pouvoir surcharger l'opérateur crochet. C'était le seul moyen de pouvoir simuler un tableau dans un objet. Cette notion n'est pas nouvelle, le langage Pascal par exemple, utilise abondamment cette notion. Un paramètre peut être passé par valeur, ou par référence. Cela permet d'éviter d'utiliser la notion de pointeur qui n'est pas évidente à maîtriser rapidement.

Dans ce paragraphe, je vais traiter des différentes situations dans lesquelles une référence peut être utilisée :

- comme attribut,

- comme paramètre d'une fonction ou d'une méthode,
- comme retour d'une fonction ou d'une méthode.

a. Attribut

Peut-on utiliser une référence comme attribut d'une classe ? La réponse est « oui », mais cela est très pénalisant. En effet, il n'est pas possible après la construction d'une référence de la modifier pour référencer un autre objet.

```
int a=0,b=1;
int& rint=a;
rint=b;                                    // a=b
```

L'affectation sur une référence modifie l'objet référencé, pas la référence elle-même. La construction et l'affectation ont des comportements différents pour les références. Cela entraîne qu'il n'est pas possible d'écrire un opérateur d'affectation sur une classe possédant un attribut en référence. L'opérateur par défaut n'est d'ailleurs pas généré par le compilateur pour les classes ayant un attribut en référence.

```
class CEmploye
{ CEntreprise& _entreprise;
  public:
    CEmploye(CEntreprise& x)
  : _entreprise(x) { }
};
```

Le compilateur ne génère pas l'opérateur d'affectation. Il ne faut donc pas utiliser les références comme attribut d'un objet.

b. Paramètres

L'utilisation d'une référence en paramètre est une question plus difficile à trancher. Il existe plusieurs sémantiques possibles pour un paramètre. Suivant les cas, les références sont à privilégier ou à bannir.

Consultation

Si le paramètre ne doit être que consulté par la méthode, il faut recevoir une référence. Si vous avez la tentation de recevoir un pointeur constant qui ne doit pas être adopté, utilisez une référence constante. En effet, cela permet à l'utilisateur de construire un objet temporaire lors de l'appel de la méthode.

```
void f(const CPersonne& personne);
```

```
void main()
{ f(CPersonne("Paul"));
}
```

Si la fonction `f()` recevait un pointeur, l'utilisateur devrait écrire :

```
void f(const CPersonne* personne);

void main()
{ CPersonne paul("Paul");
  f(&paul);
}
```

car une écriture comme `f(&CPersonne("paul"))` est refusée par le compilateur. De plus, le compilateur peut lui-même construire l'objet temporaire lors d'une écriture comme cela :

```
f("paul");                                  // Creation d'obj temporaire
```

Il est alors possible d'utiliser une conversion par construction. Un constructeur ne recevant qu'un seul paramètre est une conversion par construction. Par exemple, s'il existe une classe `CDieu` n'héritant pas de `CPersonne`, et que la classe `CPersonne` accepte un constructeur recevant comme paramètre un objet `CDieu` :

```
class CDieu
{ //...
};
class CPersonne
{ //...
  public:
    CPersonne();
    CPersonne(const CDieu& dieu);
};
```

La fonction `f()` peut directement recevoir un `CDieu` si elle attend un paramètre en référence.

```
void f(const CPersonne&);
void main()
{ CDieu zeus;
  f(zeus);
}
```

Si la fonction `f()` recevait un pointeur, il faudrait compliquer inutilement l'appel.

```
void f(const CPersonne*);
void main()
{ CDieu zeus;
  CPersonne tmp(zeus);
  f(&tmp);
}
```

Il est préférable de laisser le compilateur faire ce travail à votre place. Votre interface sera alors beaucoup plus riche. Sans le décrire directement, la fonction `f()` peut recevoir une `CPersonne` ou un `CDieu`, ou tout objet ayant une conversion vers le type `CPersonne`.

NULL possible

Si le paramètre qu'une méthode reçoit peut ne pas être présent, c'est-à-dire que l'utilisateur peut envoyer la valeur `NULL`, il faut recevoir un pointeur. Il ne faut pas utiliser une référence artificielle sur l'adresse `NULL` ce qui permettrait à la méthode d'écrire

```
void f(const A& x)
{ if (&x==NULL) ...                          // A ne pas faire !
}
```

Il faut recevoir un pointeur à la place. Une meilleure approche est de surcharger les méthodes. Une version reçoit un objet, et une autre reçoit un pointeur qui doit toujours être `NULL`.

```
void f(const A& x)
{ ...
}
void f(const void* x)
{ assert(x==NULL);
  ...
}
```

Objet Null possible

Un objet `NULL` est une instance particulière de la classe permettant de signaler qu'un objet est erroné (Voir « Instance NULL », page 85). Dans ce cas, le pointeur `NULL` n'a plus de raison d'exister. Il faut alors recevoir un paramètre du type référence si la méthode ne fait que consulter l'objet.

Adoption

Si le paramètre doit être adopté par la classe, il faut le recevoir en pointeur. Un paramètre est adopté par une classe, si celle-ci doit s'occuper de détruire l'objet en paramètre. La

destruction s'effectuant par un `delete`, l'objet doit avoir été construit par un `new`. Une précondition peut d'ailleurs vérifier que le paramètre est bien alloué dans le tas à l'aide des outils de debug mémoire (cf. page 130). L'utilisateur de l'objet manipule déjà un pointeur. Recevoir une référence l'obligerait à traduire ce pointeur en objet. De plus, l'objet receveur devrait écrire quelque chose comme `delete &x`, ce qui n'est pas très élégant, et entraînerait l'existence d'une référence invalide. Il faut donc recevoir un pointeur.

Tableau

Si le paramètre reçu est un tableau d'objet, il faut utiliser un pointeur. Les références ne permettent pas de manipuler un tableau. Recevoir un `const char*` équivaut à recevoir un tableau de caractères. Dans ce cas, il faut garder le pointeur.

```
void f(const char* p);
```

Une écriture plus claire serait par ailleurs

```
void f(const char p[]);
```

Modification

Si le paramètre doit être modifié par la méthode, il est possible de recevoir un pointeur ou une référence.

```
void f(CPersonne* personne)
{ personne->setNom("Philippe");
}
```

ou

```
void f(CPersonne& personne)
{ personne.setNom("Philippe");
}
```

Si le paramètre est reçu comme référence, l'utilisateur de la méthode peut être tenté de construire un objet temporaire. Cet objet serait modifié juste avant d'être détruit. Cela ne sert à rien. C'est d'ailleurs pour cela que le compilateur ne génère pas d'objet temporaire pour alimenter une référence non constante.

```
f("paul");            // ou
f(CPersonne("paul")); // warning ou erreur du compilateur
```

Il est donc préférable d'utiliser dans ce cas un pointeur. Cela signale à l'utilisateur que l'objet pointé sera modifié par la méthode et évite la construction hors norme d'un objet temporaire pour modification.

c. Return

Variable ou paramètre

Il n'est pas possible de retourner une référence ou un pointeur sur une variable ou un paramètre d'une méthode (Voir la règle CPP.DEB.15, page 186).

Retourner une agrégation

Il faut bien identifier si l'attribut représente une agrégation ou une relation. Pour retourner un attribut, utilisez une référence constante. Un pointeur sur un attribut a une durée de vie qui n'est pas garantie. Si l'utilisateur de la méthode garde celui-ci alors que l'objet modifie l'emplacement de son attribut, le pointeur gardé par l'utilisateur deviendrait invalide.

```
class CEntreprise
{ CPersonne* _gerant;
  public:
    CEntreprise(const CPersonne& gerant)
  : _gerant(new CPersonne(gerant))
  { }
    ~CEntreprise()
  { delete _gerant;
  }
  const CPersonne* getGerant() const
  { return _gerant; }
  void chgGerant(const CPersonne& gerant)
  { delete _gerant;
    _gerant=new CPersonne(gerant);
  }
};
void main()
{ CEntreprise WorldCompagnie(CPersonne("paul"));
  const CPersonne* gerant=WorldCompagnie.getGerant();
  WorldCompagnie.chgGerant(CPersonne("pierre"));
  // Le pointeur "gerant" n'est plus valide !
}
```

Si vous retournez une référence, l'utilisateur n'est pas tenté de garder le pointeur sur l'attribut.

```
class CEntreprise
{ CPersonne* _gerant;
  public:
    CEntreprise(const CPersonne& gerant)
```

```
  : _gerant(new CPersonne(gerant))
  { }
    ~CEntreprise()
  { delete _gerant;
  }
  const CPersonne& getGerant() const
  { return *_gerant; }
  void chgGerant(const CPersonne& gerant)
  { delete _gerant;
    _gerant=new CPersonne(gerant);
  }
};
```

Manipuler une référence est plus difficile que de manipuler un pointeur. La durée de vie d'une référence est en général courte, car il n'est pas possible de changer la référence. Seul l'objet référencé peut être modifié. Ne pouvant pas être un attribut d'une classe, seule une fonction peut manipuler la référence. Le risque de pointeur erroné est alors réduit.

Retourner une agrégation avec NULL possible

Si un pointeur représente une agrégation et que ce pointeur peut posséder la valeur `NULL`, ce pointeur représente une agrégation optionnelle. Dans ce cas, il faut retourner un pointeur constant pour pouvoir transmettre l'information de l'absence de l'attribut. Il est préférable d'avoir un Objet `Null` pour la classe de l'attribut.

Retourner une agrégation avec Objet Null possible

S'il existe une instance particulière indiquant que l'objet est en erreur ou vide, retournez alors une référence constante. Il n'est plus nécessaire de retourner un pointeur qui pourrait être gardé par erreur par l'appelant.

Retourner dans un conteneur

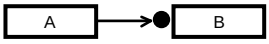
Un conteneur se contente de garder un ensemble d'objets indépendamment des objets eux-mêmes. L'accès à un des objets du conteneur doit permettre de modifier l'objet directement dans le conteneur. Dans ce cas, il faut retourner une référence non constante sur les objets contenus. Un objet de type « tableau » doit retourner une référence.

```
class CTableauA
{ A _tab[10];
  public:
  A& operator[](int index)
  { return _tab[index]; }
};
```

Cela permet de modifier les objets A directement dans l'objet CTableauA. Même si les objets A sont des agrégations pour l'objet CTableauA, il faut offrir l'accès direct à ces attributs car le tableau ne fait que les regrouper, mais est indépendant des traitements sur les objets contenus.

Retourner une relation

Une relation indique par principe que plusieurs objets peuvent pointer sur un objet en relation. Si l'objet pointé doit être détruit dans le destructeur de la classe, ce serait une agrégation, donc un attribut. Pour un objet A en relation avec un objet B, l'objet A n'étant pas propriétaire de l'objet B, toute modification de l'objet B est valide.



De plus sa durée de vie n'est pas dépendante de l'objet A. Dans ce cas, il faut retourner un pointeur non constant. Modifier l'objet B n'entraîne pas, sémantiquement, la modification de l'objet A.

```
class A
{ B* _relaB;
  public:
  B* getRelaB() const
  { return _relaB; }
};
```

Retourner this

Retourner this permet à l'utilisateur de la méthode d'enchaîner les appels. Il faut dans ce cas retourner une référence afin de pouvoir utiliser directement les opérateurs de la classe. Une écriture comme suit :

```
a=b=c;
```

n'est valide que si l'opérateur d'affectation retourne une référence sur lui-même.

```
A& A::operator =(const A& x)
{ //...
  return *this;
}
```

Une méthode constante doit retourner une référence constante, une méthode non-constante doit retourner une référence non constante.

d. Résumé

Voici un tableau récapitulatif des utilisations des références.

		Tableau	NULL Possible	NULL impossible	Objet NULL
Déclarer un attribut	!const	*	*	*	*
	const	*	*	*	*
Recevoir un paramètre	!const	*	*	*	&
	const	*	* &	&	&
Retourner un attribut	!const	*	*	&	&
	const	*	*	&	&
Retourner une relation	!const	*	*	*	*
	const	*	*	*	*
Retourner un conteneur	!const	/	/	&	&
	const	/	/	&	&
Retourner this	!const	/	/	&	/
	const	/	/	&	/

/ : hors sujet * : pointeur & : référence

- Il est rare de recevoir une référence non `const` en paramètre.
- Un paramètre pointeur constant représente toujours un tableau.

C. POURQUOI UTILISER <IOSTREAM.H> ?

Les flux offerts par le C++ sont un service dont on peut se passer. Les programmeurs C utilisent `printf` et connaissent parfaitement son utilisation. Pourquoi changer ses habitudes lorsque cela fonctionne parfaitement ?

Bjarne Stroustrup avait lu dans un article qu'il ne serait pas possible, avec un langage, de construire par programme un mécanisme d'écriture dans un flux efficace et optimisé. Le C interprète une chaîne de caractères pour afficher les objets. Le pascal ou le basic ont une syntaxe particulière dans le langage pour gérer cela. Cet article a été pris par [Stroustrup, DE:94] comme un défi. Il a modélisé les premiers flux, dans ce but. Jerry Schwarz a repris son travail pour l'optimiser. Les flux sont un des meilleurs exemples d'utilisation du langage.

Pourquoi utiliser les flux ?

- Le compilateur connaît le type des objets à afficher, le contrôle de validité est effectué dès la compilation.

- `printf` est interprété alors que les flux C++ sont compilés. Des fonctions pertinentes et `inline` sont générées par le compilateur pour chaque type d'objet.
- Il est aisé d'ajouter de nouveaux types d'affichages selon les objets, ce qui est impossible avec `printf` (Format de date ou autre...).
- Les flux C++ sont de vrais objets. Il est possible d'écrire d'autres flux sans modifier les fonctions d'affichage. Il n'y a pas plusieurs fonctions suivant l'objectif du flux (`printf`, `sprintf`, `fprintf`, `vprintf`, ...). Il est très facile de rédiger un nouveau flux écrivant dans une fenêtre, ou dans un « pipe » par exemple. Il n'est pas nécessaire de modifier les routines d'affichage pour utiliser ce nouveau flux.

a. Comment afficher une classe

```
class CFraq
{ public:
  friend ostream& operator <<(ostream& o,const CFraq& x)
  { return o << x._num << '/' << x._den; }
  // ...
  private:
  int _num,_den;
};
```

Avec cette seule fonction `inline`, il est possible d'écrire l'objet dans un tampon, dans un fichier, à l'écran, et plus tard, dans un pipe, un modem, ou une fenêtre sans avoir à modifier l'objet `CFraq`. Tous les flux présents et futurs sont compatibles avec l'opérateur « `<<` ». Tout objet devrait avoir ce type d'opérateur, ne serait-ce que pour le débuggage afin de pouvoir tracer l'objet dans une fenêtre ou un fichier.

b. Affichage polymorphe

Si une classe est polymorphe, c'est-à-dire qu'il existe des classes dérivées, il faut offrir un mécanisme d'affichage polymorphe. Une méthode virtuelle doit être ajoutée à la classe de base qui sera appelée par l'opérateur de flux de celle-ci.

```
class CBase
{ public:
  virtual ostream& print(ostream& o) const =0;
  friend inline ostream& operator <<(ostream& o,const CBase& x)
  { return x->print(o); }
};

class CDerive : public CBase
{ ostream& print(ostream& o) const
  { return o << ...;
```

```
  }
};
```

Pour apporter une amélioration aux flux, consultez le chapitre « Indentation des flux », page 123.

D. IDENTIFICATION D'INSTANCES

Un objet est un élément indépendant. Deux objets ayant les mêmes valeurs sont considérés comme des objets différents. Un objet est identifié par son existence. En C++ l'identification d'un objet est son adresse mémoire.

Il arrive fréquemment qu'une classe désire identifier différemment chaque instance. Par exemple, un homme peut être identifié par son numéro de sécurité sociale. Il ne faut en aucun cas avoir deux personnes ayant le même numéro. C'est typiquement le type de champs devant être indexé dans une base de données. Cela pose un certain nombre de contraintes sur les objets. En effet, une instance identifiée ne peut plus être recopiée. Le constructeur de copie et l'opérateur d'affectation ne peuvent pas être écrits. Il n'est pas possible de construire une identification arbitraire pour un nouvel objet. On ne va pas inventer un code de sécurité sociale pour identifier la copie ! Il faut alors déclarer le constructeur de copie et l'opérateur d'affectation en `protected`.

```
class CHomme
{ long _numsecu;
  public:
    CHomme(long numsecu) : _numsecu(numsecu) {}
  protected:
    CHomme(const CHomme&);                  // Non implementé
  void operator =(const CHomme&);           // Non implementé
};
```

Il faut dans la mesure du possible, éviter d'avoir des classes ayant un identifiant. Vous ne pouvez que retourner une référence sur une instance de ce type. Il est impossible de retourner une copie.

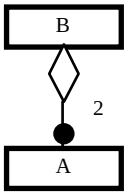
CHAPITRE 5

OUTILS DE DEBUG

A. INDENTATION DES FLUX

Pour pouvoir facilement debugger, il est d'usage d'avoir un opérateur de flux pour chaque objet.

```
class A
{ int _a;
  int _b;
  public:
    A(int a,int b) : _a(a), _b(b) {}
  friend ostream& operator <<(ostream& o,const A& x)
  { o << "A:" << endl
      << "{ _a=" << x._a << endl
      << "  _b=" << x._b << endl
      << "}" << endl;
    return o;
  }
};
```



```
class B
{ A _a1;
  A _a2;
  public:
    B(A a1,A a2) : _a1(a1), _a2(a2) {}
  friend ostream& operator <<(ostream& o,const B& x)
  { o << "B:" << endl
      << "{ _a1=" << x._a1 << endl
      << "  _a2=" << x._a2 << endl
      << "}" << endl;
    return o;
  }
};
```

L'affichage d'un objet `B` est du format suivant :

```
B:
{ _a1=A:
{ _a=1
  _b=2
}

  _a2=A:
{ _a=3
  _b=4
}

}
```

Cette trace est difficile à lire car il n'y a pas d'indentation lors de l'affichage des attributs `_a1` et `_a2` de l'objet `B`. Il est souhaitable d'avoir un mécanisme pour pouvoir indenter, comme les sources C++, les traces des objets les uns dans les autres. Pour cela, nous allons ajouter cette fonctionnalité aux flux.

Il faut maintenir un paramètre supplémentaire pour connaître l'indentation courante d'un flux. Ceux-ci offrent un moyen d'ajouter un entier dans chacun. Pour cela, il faut appeler la méthode statique `ios::xalloc()`. Celle-ci renvoie un entier servant d'index à la méthode `ios::iword()` afin de pouvoir maintenir un entier dans un flux.

Nous allons utiliser cette méthode pour connaître l'index de l'entier à ajouter.

```
int iostream_index_indent=ios::xalloc();
```

Nous utiliserons ensuite la valeur de `iostream_index_indent` pour manipuler l'indentation courante.

Il faut ensuite ajouter deux manipulateurs pour incrémenter ou décrémenter l'indentation courante, et un manipulateur de remise à zéro.

```
inline ostream& incindent(ostream& o)
{ ++o.iword(iostream_index_indent);
  return o;
}

inline ostream& decindent(ostream& o)
{ if (o.iword(iostream_index_indent))
    --o.iword(iostream_index_indent);
  return o;
}
inline ostream& resetindent(ostream& o)
{ o.iword(iostream_index_indent)=0;
  return o;
}
```

La valeur entière présente dans chaque flux à l'index `iostream_index_indent` n'est pas initialisée à zéro. Il faut alors appeler le manipulateur `resetindent` avant d'utiliser l'indendation pour un flux. Pour pouvoir initialiser correctement les flux standard `cout`, `cerr` et `clog` avant le `main`, nous allons ajouter une instance globale d'initialisation.

```
static struct CInitStdStream
{ CInitStdStream()
  { cout << resetindent;
    cerr << resetindent;
    clog << resetindent;
  }
} InitStdStream;
```

L'instance globale `InitStdStream` permet d'initialiser correctement les flux standard.

Il faut ensuite, déclarer un nouveau manipulateur pour gérer l'indentation. Nous allons l'appeler `iendl`, comme « Indentation `endl` ».

```
ostream& iendl(ostream& o)
{ o << '\n';
  int mem=o.width(o.iword(iostream_index_indent)*2);
  o << "" << flush;
  o.width(mem);
  return o;
}
```

Avec ces outils, l'utilisateur peut écrire :

```
{ cout << "abc" << incindent << iendl
       << "def" << decindent << iendl
       << "ghi" << iendl;
}
```

pour obtenir :

```
abc
  def
ghi
```

Pour simplifier l'utilisation de cet outil, ajoutons deux manipulateurs supplémentaires, permettant de modifier le niveau d'indentation en retournant à la ligne.

```
inline ostream& incendl(ostream& o)
{ return o << incindent << iendl; }
inline ostream& decendl(ostream& o)
{ return o << decindent << iendl; }
```

L'utilisateur peut modifier le premier exemple comme cela :

```
friend ostream& operator <<(ostream& o,const A& x)
{ return o << "A:"            << incendl
           << "{ _a=" << x._a << iendl
           << "  _b=" << x._b << iendl
           << '}'             << decindent;
}
friend ostream& operator <<(ostream& o,const B& x)
{ return o << "B:"              << incendl
           << "{ _a1=" << x._a1 << iendl
           << "  _a2=" << x._a2 << iendl
           << '}'               << decindent;
}
```

afin d'obtenir à l'écran :

```
B:
  { _a1=A:
    { _a=1
      _b=2
    }
    _a2=A:
    { _a=3
      _b=4
    }
  }
```

Ce qui est plus lisible. Toute la hiérarchie des objets va pouvoir apparaître. Par commodité, il est recommandé d'ajouter une vérification de l'adresse de l'objet.

```
friend ostream& operator <<(ostream& o,const A& x)
{ if (&x==NULL) return o << "A:NULL";
  return o << "A:"            << incendl
           << "{ _a=" << x._a << iendl
           << "  _b=" << x._b << iendl
           << '}'             << decindent;
}
```

Il arrive en effet fréquemment, que l'utilisateur désire afficher un objet à partir de son adresse.

```
void f(A* p)
{ cout << *p;
}
```

L'ajout de la vérification de l'adresse de l'objet, permet d'afficher la valeur NULL. Cela est très pratique lors de l'affichage d'attributs.

Pour montrer l'héritage il faut rédiger les opérateurs de flux de chaque classe comme suit :

```
class A
{ int _a;
  public:
  friend ostream& operator <<(ostream& o,const A& a);
};

inline ostream& operator <<(ostream& o,const A& a)
{ return o << "A:"            << incendl
           << "{ _a=" << a._a << iendl
           << '}'             << decindent;
}

class B : public A
{ int _b;
  public:
  friend ostream& operator <<(ostream& o,const B& b);
};

inline ostream& operator <<(ostream& o,const B& b)
{ return o << "B:"           << incendl
           << "{ " << (A&)b  << iendl
           << "  _b=" << b._b << iendl
           << '}'            << decindent;
}
```

Cela affichera :

```
b=B:
  { A:
    { _a=0;
    }
    _b=0;
  }
```

L'héritage s'affiche comme un attribut mais sans nom.

Il est utile de pouvoir afficher un objet dérivé à partir d'un objet de base. Par exemple, avec les classes A et B précédentes, si l'utilisateur écrit :

```
void main()
{ B b;
  A* pa=&b;
  cout << *pa;
}
```

La trace affichera un objet A et non un objet B. L'affichage dépendra de type du pointeur et non du type de l'objet pointé.

Pour pouvoir afficher l'objet B à partir d'un pointeur A, il faut ajouter une méthode `virtual` dans la classe A,

```
class A
{ int _a;
  public:
  friend ostream& operator <<(ostream& o,const A& a);
  virtual ostream& print(ostream& o) const
  { return o << "A:"          << incendl
             << "{ _a=" << _a << iendl
             << '}'           << decindent;
  }
};
```

et déclarer l'opérateur de flux pour cette classe.

```
inline ostream& operator <<(ostream& o,const A& x)
{ if (&x==NULL) return o << "A:NULL";
  return x.print(o);
}
```

Il faut également ajouter un nouvel opérateur de flux.

```
inline ostream& operator <<(ostream& o,ostream&)
{ return o; }
```

Pour la classe dérivée B, il faut redéclarer la méthode `print`.

```
class B : public A
{ int _b;
  public:
  friend ostream& operator <<(ostream& o,const B& b);
  virtual ostream& print(ostream& o) const
  { return o << "B:"                 << incendl
             << "{ " << A::print(o) << iendl
             << "  _b=" << _b        << iendl
             << '}'                 << decindent;
  }
};
```

L'opérateur de flux pour un objet `ostream` permet de ne pas afficher l'adresse du flux `o` lors de l'appel de `A::print(o)`.

Avec ces outils il est possible d'afficher un objet pointé par un pointeur de la classe de base et d'avoir l'objet dérivé correspondant.

```
void main()
{ B b;
  A* pa=&b;
  cout << "*pa=" << *pa << endl;
}
```

affiche

```
*pa=B:
  { A:
    { _a=0;
    }
    _b=0;
  }
```

Le fichier "`indent.h`" est le suivant :

```
#include <iostream.h>

extern int iostream_index_indent;

ostream& iendl(ostream& o);

inline ostream& incindent(ostream& o)
{ ++o.iword(iostream_index_indent);
  return o;
}
inline ostream& decindent(ostream& o)
{ if (o.iword(iostream_index_indent))
    --o.iword(iostream_index_indent);
  return o;
}
inline ostream& resetindent(ostream& o)
{ o.iword(iostream_index_indent)=0;
  return o;
}
inline ostream& incendl(ostream& o)
{ return o << incindent << iendl; }
inline ostream& decendl(ostream& o)
{ return o << decindent << iendl; }
inline ostream& operator <<(ostream& o,ostream&)
{ return o; }
```

Le fichier "`indent.cpp`" est le suivant :

```
#include "indent.h"

int iostream_index_indent=ios::xalloc();

ostream& iendl(ostream& o)
{ o << '\n';
  int mem=o.width(o.iword(iostream_index_indent)*2);
  o << "";
  o.width(mem);
  return o;
}
static struct CInitStdStream
{ CInitStdStream()
  { cout << resetindent;
    cerr << resetindent;
    clog << resetindent;
  }
} InitStdStream;
```

Avec ces quelques lignes, on ajoute une nouvelle fonctionnalité aux flux. Ceux-ci ont été suffisamment bien étudiés pour offrir cette possibilité.

B. `::new` DE DEBUG

Le langage C++ permet de redéfinir les opérateurs `::new` et `::delete`. Cela permet d'écrire une version de debug vérifiant leur utilisation. Les différents problèmes mémoire pouvant arriver sont les suivants :

- effacement d'une zone mémoire non allouée précédemment,
- effacement d'une zone déjà libérée,
- utilisation d'une zone mémoire libérée,
- écriture en dehors d'une zone allouée,
- zone mémoire jamais libérée.

Tous ces problèmes peuvent être partiellement détectés à l'exécution. Pour cela, il faut modifier les opérateurs d'allocation. La nouvelle version va garder une liste chaînée des zones allouées. Cela permet de vérifier si, lors d'un effacement, le pointeur fourni par l'utilisateur fait bien partie des pointeurs retournés lors d'un précédent `::new`. D'autre part, la nouvelle version va remplir les zones mémoires à effacer avec une valeur donnée pour que l'utilisation de celles-ci devienne erronée.

La détection de l'écriture en dehors d'une zone allouée est plus difficile à réaliser. En général, l'utilisateur déborde *faiblement* de la mémoire demandée. La nouvelle version va ajouter quelques octets à la fin de chaque zone mémoire, et va remplir cette mémoire supplémentaire avec une valeur particulière. Cela permet de vérifier que cette zone contient toujours les bonnes valeurs.

La mémoire perdue est un problème classique, difficile à détecter. Au fur et à mesure de l'utilisation du programme, la mémoire de l'ordinateur est consommée jusqu'à ce que plus rien ne soit disponible. Le programme peut fonctionner pendant des heures, puis s'arrêter sans raison apparente. La nouvelle version va permettre de détecter, lors de la sortie du programme, l'ensemble des allocations non effacées. La difficulté, lorsque l'on détecte cela, est de retrouver d'où les allocations ont été effectuées. La nouvelle version indiquera le nom du fichier et la ligne à laquelle les allocations ont été faites.

Un autre problème difficile à tester, est le comportement d'un programme dans un environnement critique. Comment réagit ce programme lorsque l'ordinateur n'a plus de mémoire ? La difficulté à tester cette situation provient du fait que les outils de debug sont, eux aussi, limités. La nouvelle version permettra d'indiquer la taille mémoire maximum à fournir à l'application avant de renvoyer la valeur `NULL`. Cela permet de tester le programme dans un environnement non critique.

Pour écrire ce qui précède, il faut, lors de chaque allocation, ajouter un certain nombre d'informations supplémentaires ; garder une liste des allocations, le nom du fichier ayant fait l'allocation, ainsi que la ligne concernée. Nous ajoutons un « en-tête » devant chaque allocation.

```
struct CHeadAlloc
{ unsigned    id;        // Identification de l'allocation
  const char* fileName;  // Nom du fichier ayant fait l'allocation
  unsigned    lineNum;   // Ligne du fichier ayant fait l'allocation
  unsigned    size;      // Taille du bloc alloué
  CHeadAlloc* prev;      // Bloc mémoire précédent
  CHeadAlloc* next;      // Bloc mémoire suivant
};
```

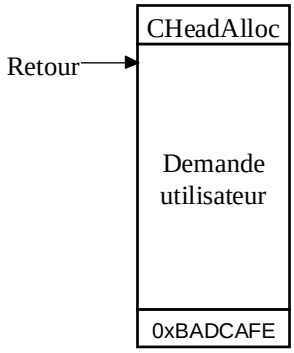
C'est incompatible avec le `malloc` du C. Un `new` ne peut pas être effacé avec un `free()`, ce que rejette d'ailleurs la norme.

On ajoutera ensuite, à la fin du bloc demandé, un `long`, ayant la valeur `0xBADCAFEL`. Cette valeur est arbitraire, elle a été choisie car elle est lisible en clair malgré la valeur hexadécimale (Bad cafe). Ceci permettra de détecter le débordement de l'utilisation du bloc. Si la fin de celui-ci n'a plus cette valeur, alors l'utilisateur a débordé.

Allouons `sizeof(CHeadAlloc) + size + sizeof(long)` octets pour chaque allocation. `size` étant la demande de l'utilisateur. Nous retournerons à l'utilisateur

l'adresse `pt + sizeof(CHeadAlloc)` et maintiendrons une liste double chaînée sur toutes les allocations.

Chaque bloc mémoire demandé est formaté comme cela :



Une difficulté à résoudre est la possibilité de connaître le nom du fichier et le numéro de la ligne pour toute allocation. Pour y parvenir on peut ajouter des paramètres à l'opérateur `::new`. Pour déclarer cet opérateur comme suit :

```
void* operator new(size_t size,const char* filename,
                   unsigned linenum);
```

l'utilisateur devra alors utiliser :

```
new (__FILE__,__LINE__) char[10];
```

Pour éviter cette écriture inconfortable, on déclarera une macro :

```
#define DEBUG_NEW new(__FILE__,__LINE__)
```

afin de pouvoir redéfinir l'opérateur `new` par « `#define new DEBUG_NEW` ». Dès lors, toute utilisation classique de l'opérateur `new` ajoutera automatiquement le nom du fichier et la ligne ayant effectué l'allocation.

Malheureusement, cette astuce est incompatible avec certaines écritures. Si un utilisateur déclare une classe ayant son propre opérateur `new`, la macro précédente entraînera une écriture syntaxiquement erronée :

```
class A
{ void* operator new();
```

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

entraînera pour le compilateur :

```
class A
{ void * operator new(__FILE__,__LINE__)();
  // ...
};
```

Ce qui n'est évidemment pas acceptable pour le compilateur. Il faut à nouveau trouver une solution à ce problème.

Pour cela, ajoutons plusieurs macros permettant de détecter l'utilisation des nouveaux opérateurs de debug. De plus, l'opérateur `new` de la classe devra pouvoir recevoir les deux paramètres supplémentaires.

```
#define DEF_NEW
#define new DEBUG_NEW
// ...
class A
{
  #ifdef DEF_NEW
  #undef new
  void* operator new(size_t size,const char* filename,
                     unsigned linenum);
  #define new DEBUG_NEW
  #else
  void* operator new(size_t size);
  #endif
};
```

Ce cas est suffisamment rare, pour ne pas poser de problèmes particuliers.

Les opérateurs `::new` et `::delete` sont déclarés pour l'ensemble du programme. Si l'on désire tester les allocations faites par les librairies déjà compilées, la signature de l'opérateur `::new` reste classique. Il n'y a pas de paramètre supplémentaire. Il faut alors ajouter :

```
void* operator new(size_t size) { return operator new (size,NULL,0); }
```

pour pouvoir identifier l'utilisation de la mémoire. Il faut de plus, ajouter un identifiant incrémenté à chaque allocation. Ceci permet de connaître le numéro de chaque allocation, et de remonter ainsi à la ligne concernée du source.

Pour pouvoir tester le programme dans les cas limites, il faut offrir à l'utilisateur le moyen d'indiquer la taille mémoire maximum autorisée lors des allocations. A cette fin, on ajoutera une variable globale indiquant ce paramètre. Lors de chaque allocation, la taille courante

est maintenue afin de pouvoir la comparer avec la valeur de ce paramètre. Si une allocation dépasse cette valeur, l'opérateur retournera la valeur `NULL`. Nous utiliserons la variable `CDNew::useAllocMax`.

Par ailleurs, il peut être intéressant de renvoyer la valeur `NULL`, non pas lorsqu'une certaine quantité de mémoire est allouée, mais lorsqu'un certain nombre d'allocations est effectué ; cela, quelque soit la taille respective de chaque bloc. Nous déclarerons pour ce faire, la variable globale `CDNew::useNbMaxAlloc`.

Afin de faciliter le débuggage, nous allons également offrir des services qui permettront à l'utilisateur de connaître l'état du tas.

La variable `CDNew::nbAlloc` indique le nombre courant d'allocations mémoire. La variable `CDNew::maxAlloc` indique le nombre maximum d'octets demandés dans le tas qui permet de connaître les ressources maximum nécessaires à l'application. La variable `CDNew::allocCur` indique le nombre courant d'octets demandés dans le tas. Celui-ci permet de vérifier si la mémoire utilisée après l'appel d'une fonction est identique à celle avant son appel.

```
#ifdef DEF_NEW
int memalloc=CDNew::allocCur;
#endif
f();                                          // Appel de la fonction
#ifdef DEF_NEW
assert(memalloc== CDNew::allocCur);
#endif
```

La fonction `CDNew::dump()` affiche sur le flux indiqué, l'ensemble des allocations du tas. Cette fonction appelée lors de la destruction d'un objet statique présent dans la librairie permet d'afficher les allocations restantes lors de la fin du programme.

Il y a ici un problème insoluble. Il n'est pas possible de modifier l'ordre d'initialisation des objets statiques (Voir règle CPP.DEB.27, page 200). L'objet appelant `CDNew::dump()` doit être appelé en tout dernier, sinon il peut rester dans le tas des allocations faites par les objets statiques n'étant pas encore détruits. Certains compilateurs permettent d'ordonner cela. Ce n'est pas toujours le cas. Les flux standard sont typiquement des objets statiques allouant de la mémoire sur le tas, mais détruits après notre objet statique. Pour réduire les traces de ces allocations, la fonction `CDNew::dump()` peut recevoir un paramètre, indiquant de ne tracer que les allocations ayant été effectuées avec un nom de fichier. Les librairies n'étant pas concernées, cela permet de ne tenir compte que des allocations faites *via* l'utilisation classique, avec nom, de notre outil.

L'opérateur `::new` appelle une fonction paramètre avec `set_new_handler()` lorsqu'il n'y a plus de mémoire. Si cette fonction n'est pas déclarée, la valeur `NULL` est renvoyée. La difficulté est de connaître la valeur courante de cette fonction. Le seul moyen portable est de modifier cette valeur. La fonction `set_new_handler()` retourne alors

la valeur précédente. Il faut remettre celle-ci en place, et appeler la fonction récupérée si l'allocation échoue. Cela s'effectue à l'aide des lignes suivantes :

```
{ void (*CurrentNewHandle)()=set_new_handler((void(*)())NULL);
  if (CurrentNewHandle!=NULL)
  { set_new_handler(CurrentNewHandle);
    CurrentNewHandle();
  }
  else
    return NULL;
}
```

L'utilisation de cet outil est très simple. Il suffit d'inclure le fichier "dnew.h" avant la première allocation.

Pour mémoriser le nom du fichier et la ligne ayant fait le new, il faut ajouter :

```
#define DEF_NEW
#define new DEBUG_NEW
```

A partir de ce moment, toutes les allocations du fichier, mémorisent les lignes où elles ont été effectuées.

Voici le source du fichier dnew.h :

```
// Lors du #define new DEBUG_NEW, ajoutez #define DEF_NEW

#ifndef DNEW_H
#define DNEW_H

#ifndef NDEBUG
#define DNEW_FILE __FILE__
void* operator new(size_t size,const char*filename,unsigned linenum);
#define DEBUG_NEW new(DNEW_FILE,__LINE__)

class CDNew
{ public:
  static void dump(ostream& o,bool mode=true);
  static unsigned             useNbMaxAlloc; // Nb alloc autorisées
  static unsigned long        useMaxAlloc;   // Alloc. max autorisé
  static const unsigned&      nbAlloc;       // Nb alloc en cours
  static const unsigned long& maxAlloc;      // Nb max d'alloc
  static const unsigned long& allocCur;      // Allocation courante
};
#else // NDEBUG
  #define DEBUG_NEW new
#endif // NDEBUG
#endif // DNEW_H
```

Et celui de `dnew.cxx` :

```
#ifndef NDEBUG
#include <stdlib.h>
#include <string.h>
#include <new.h>
#include <iostream.h>
#include <stdio.h>
#include <assert.h>

#include "dnew.h"

struct CHeadAlloc
{ unsigned long id;                  // Identifiant de l'allocation
  const char*   fileName;            // Nom du fichier ayant alloué
  unsigned      lineNum;             // Ligne du fichier en question
  unsigned      size;                // Taille du bloc
  CHeadAlloc*   prev;                // Bloc précédent
  CHeadAlloc*   next;                // Bloc suivant
};

static struct
{ unsigned long id;                  // Identifiant suivant
  CHeadAlloc*   first;               // Premier de la chaîne
  bool          informe;             // Flag pour signaler qu'1 fois
                                     // le manque de mémoire
  unsigned nbAlloc;                  // Nb alloc en cours
  unsigned long maxAlloc;            // Nb max d'alloc
  unsigned long allocCur;            // Allocation courante
} StatAlloc={0,NULL,false,0,0,0};

const unsigned&      CDNew::nbAlloc=StatAlloc.nbAlloc;
const unsigned long& CDNew::maxAlloc=StatAlloc.maxAlloc;
const unsigned long& CDNew::allocCur=StatAlloc.allocCur;

unsigned      CDNew::useNbMaxAlloc=0; // Nb alloc max autorisées
unsigned long CDNew::useMaxAlloc=0;   // Allocations max autorisées

const long magicbound=0xBADCAFEL;

bool returnNULL(const char* filename,unsigned linenum)
{
  if (!StatAlloc.informe)
  { StatAlloc.informe=true;
    fputs("CDNew::",stderr);
    if (filename!=NULL) fprintf(stderr,"%s(%u):",filename,linenum);
    fputs("Plus de memoire !\n",stderr);
  };
  // Recupere le new_handle courant
  void(*CurrentNewHandle)()=set_new_handler((void(*)())NULL);
  if (CurrentNewHandle!=NULL)
  { set_new_handler(CurrentNewHandle);
    CurrentNewHandle();
  }
  else
```

```
    return true;
  return false;
}

void* operator new(size_t size)
{ return operator new (size,NULL,0); }

void* operator new(size_t size,const char* filename,unsigned linenum)
{
  char*  pt;
  size_t xsize;

  ++StatAlloc.nbAlloc;
  xsize=size+sizeof(CHeadAlloc)+sizeof(long);
  if ((CDNew::useMaxAlloc) &&
      (StatAlloc.allocCur+xsize>CDNew::useMaxAlloc))
  { returnNULL(filename,linenum);
    return(NULL);
  }
  do
  { pt=(char*)malloc(xsize+sizeof(CHeadAlloc)+sizeof(long));
    if ((pt==NULL) && returnNULL(filename,linenum)) return(NULL);
  } while (pt==NULL);
  ((CHeadAlloc*)pt)->id=StatAlloc.id++;
  ((CHeadAlloc*)pt)->fileName=filename;
  ((CHeadAlloc*)pt)->lineNum=linenum;
  ((CHeadAlloc*)pt)->size=size;
  ((CHeadAlloc*)pt)->prev=NULL;
  ((CHeadAlloc*)pt)->next=StatAlloc.first;
  if (StatAlloc.first!=NULL) StatAlloc.first->prev=(CHeadAlloc*)pt;
  StatAlloc.first=(CHeadAlloc*)pt;

  memcpy(pt+xsize-sizeof(long),&magicbound,sizeof(magicbound));
  if ((CDNew::useNbMaxAlloc) && (StatAlloc.nbAlloc>CDNew::useNbMaxAlloc))
  { returnNULL(filename,linenum);
    return(NULL);
  }
  StatAlloc.allocCur+=size;
  if (StatAlloc.allocCur>StatAlloc.maxAlloc)
    StatAlloc.maxAlloc=StatAlloc.allocCur;
  return(pt+sizeof(CHeadAlloc));
}

void operator delete(void* pt)
{
  --StatAlloc.nbAlloc;
  pt=(char*)pt-sizeof(CHeadAlloc);

  for (const CHeadAlloc* p=StatAlloc.first;p!=NULL;p=p->next)
  { if ((CHeadAlloc*)p==pt) break;
  }
  if (p==NULL)
  { fputs("CDNew::Allocation inconnue !\n",stderr);
    assert(p!=NULL);                  // delete bloque sans new !
    return;
  }
```

```
  // Trap de fin de bloque
  if (memcmp(((char*)pt+sizeof(CHeadAlloc)+((CHeadAlloc*)pt)->size),
             &magicbound,sizeof(magicbound)))
  { fputs("CDNew::",stderr);
    if (((CHeadAlloc*)pt)->fileName!=NULL)
      fprintf(stderr,"%s(%u):",((CHeadAlloc*)pt)->fileName,
              ((CHeadAlloc*)pt)->lineNum);
    fputs("Debordement dans l'utilisation d'un bloc !\n",stderr);
    assert(0);
    return;
  }
  StatAlloc.allocCur-=((CHeadAlloc*)pt)->size;
  if (((CHeadAlloc*)pt)->prev!=NULL)
  { ((CHeadAlloc*)pt)->prev->next=((CHeadAlloc*)pt)->next; }
  if (((CHeadAlloc*)pt)->next!=NULL)
  { ((CHeadAlloc*)pt)->next->prev=((CHeadAlloc*)pt)->prev; }
  if (StatAlloc.first==pt)
  { StatAlloc.first=((CHeadAlloc*)pt)->next; }
  memset(pt,0xCA,((CHeadAlloc*)pt)->size); // Efface le buffer
  free(pt);
}

void CDNew::dump(ostream& o,bool mode)
{
  bool fl=false;
  for (const CHeadAlloc* p=StatAlloc.first;p!=NULL;p=p->next)
  { if ((mode) && (p->fileName==NULL)) continue;
    if (!fl)
    { o << "CDNew::Dump:" << endl;
      fl=true;
    }
    o << (void*)p << ':' << p->id << ' ';
    if (p->fileName!=NULL)
      o << p->fileName << '(' << p->lineNum << ") ";
    o << p->size << endl;
  }
}

static struct CStaticMemoryLeak
{ unsigned last;
  CStaticMemoryLeak() : last(CDNew::nbAlloc) { }
  ~CStaticMemoryLeak()
  { if (CDNew::nbAlloc>last) CDNew::dump(cerr,true);
  }
} MemoryLeak;

#endif
```

Nous allons tester cet outil à l'aide d'un exemple, où plusieurs erreurs vont être détectées.

```
1      #include <string.h>
2      #include <iostream.h>
3      #include "dnew.h"

4      #define new DEBUG_NEW
```

```
5       struct CNode
6       { CNode*      next;
7         const char* data;
8           CNode(const char* xData,CNode* xNext=NULL)
9         : data(xData), next(xNext) {}
10      };

11      void affiche_mots(char* str)
12      {
13        CNode* cur;
14        CNode* first;
15        char*  buf;
16        char*  token;

17        buf=new char [strlen(str)];
18        strcpy(buf,str);

19        // Découpage de la phrase en mots
20        token=strtok(buf," \t");
21        cur=new CNode(token);
22        while (token!=NULL)
23        { token=strtok(NULL," \t");
24          if (token!=NULL)
25            cur=new CNode(token,cur);
26        }
27        first=cur;

28        // Affichage des mots
29        for (cur=first;cur!=NULL;cur=cur->next)
30          cout << '[' << cur->data << ']' << endl;

31        // Liberation des noeuds
32        for (cur=first;cur!=NULL;cur=cur->next)
33          delete cur;

34        delete [] str;
35      }

36      const int i=3;

37      void main()
38      {
39        char* pt=new char[255];
40        strcpy(pt,"Ceci est une phrase à decouper");
41        affiche_mots(pt);
42        delete [] pt;
43      }
```

Cet exemple décompose une chaîne de caractères en mots. Chaque mot est stocké dans une liste d'objets `CNode`. Ensuite, le programme affiche chacun des mots de la phrase. Puis, toute la mémoire utilisée est effacée.

Lors de l'exécution, un `assert` est déclenché à l'exécution de la ligne trente-trois. Le programme détecte l'effacement d'un bloc non alloué.

Lors de la libération d'un bloc mémoire, toutes les données sont mises à la valeur « `0xCA` ». L'utilisation de `cur=cur->next` devient alors erronée. La valeur de `cur` devient aléatoire. Par la suite, un nouvel appel à `delete` est demandé sur une adresse incorrecte, ce qui est détecté par la librairie.

Ce type d'erreur passe généralement inaperçu. En effet, lors de la libération habituelle de la mémoire, celle-ci n'est pas effacée. Elle reste avec des valeurs valides, les dernières de l'objet. Cela peut fonctionner tant que le système d'exploitation n'a pas besoin de cette mémoire. C'est typiquement une erreur aléatoire. La plus difficile à localiser.

Il faut corriger le programme en modifiant la boucle par :

```
CNode* ocur;
for (cur=first;cur!=NULL;cur=ocur)
{ ocur=cur->next;
  delete cur;
}
```

Une nouvelle compilation puis exécution, fait apparaître une nouvelle erreur à la ligne quarante-deux du fichier original. Le pointeur `pt` ne serait pas valide. En effet, la fonction `affiche_mots()` efface à la ligne trente-quatre ce même bloc mémoire. Il faut supprimer la ligne trente-quatre du fichier.

L'exécution suivante semble fonctionner parfaitement, sauf, que le programme indique qu'une zone mémoire n'a jamais été libérée. Cette mémoire a été allouée à la ligne dix-sept du fichier. En effet, il manque la destruction de ce tampon. Il faut ajouter à la fin de la fonction `affiche_mots()`, la commande « `delete [] buf` ».

Après avoir, pour la quatrième fois, compilé le programme, vous constatez encore une erreur lors de l'effacement de `buf`. Celle-ci indique un débordement dans l'utilisation du tampon. Il faut alors regarder où l'application utilise ce pointeur. La ligne dix-huit recopie la chaîne de caractères `str`. Pour cela, il faut que le tampon possède `strlen(str)+1` caractères disponibles. Il faut ajouter une unité pour pouvoir stocker le caractère `'\0'`. On constate que la ligne précédente ne prévoit pas suffisamment de mémoire pour copier complètement la chaîne de caractères. Il faut la corriger ainsi :

```
buf=new char [strlen(str)+1];
```

Cette dernière exécution fonctionne parfaitement. Nous venons de parcourir l'ensemble des erreurs mémoire que permet de détecter cette librairie. Celle-ci est moins rapide que la version classique, mais elle permet de détecter les erreurs. Elle permet de détecter l'appel de `delete X` à la place de `delete [] X` ou autres subtilités. Il est possible d'enrichir

cette librairie en offrant par exemple une méthode permettant de savoir si un pointeur a été alloué dans le tas.

```
bool CDNew::isHeap(void* pt)
{ pt=(char*)pt-sizeof(CHeadAlloc);
  for (const CHeadAlloc* p=StatAlloc.first;p!=NULL;p=p->next)
  { if (p==pt) return (true);
  }
  return (false);
}
```

Cela permet de vérifier les préconditions d'une routine. Un objet recevant en paramètre un pointeur qu'il désire adopter, peut tester la validité de celui-ci en précondition.

```
class CString
{ const char* p;
  public:
    CString(const char* x)
  : p(x)
  { assert(CDNew::isHeap(x));              // Teste la précondition
  }
    ~CString()
  { delete [] p; }
};
```

Une construction de cet objet comme suit :

```
void main()
{ char buf[10];
  CString str(buf);                        // Erreur
}
```

sera détectée à l'exécution. Le pointeur `buf` fourni à l'objet `str` n'a pas été alloué dans le tas. Il n'est pas possible d'appeler `delete []()` pour ce pointeur. Le test de la précondition permet de vérifier la validité des paramètres le plus tôt possible. Sans ce test, l'erreur aurait été détectée lors de l'appel du destructeur de `CString`. La librairie aurait détecté un effacement incorrect.

Il est envisageable d'enrichir cet outil pour offrir par exemple :

- une trace de toutes les allocations,
- une trace de toutes les libérations,
- une trace ponctuelle sur la libération d'un objet,
- l'ajout d'une pile maintenant le numéro d'allocation courant afin de ne tracer que les objets de numéro supérieurs (`push()`, `pop()`, `CDNew::dumplast()`),

- etc.

Cette librairie ne cherche pas à rivaliser avec d'autres utilitaires plus spécialisés qui utilisent les capacités de mémoires virtuelles des microprocesseurs pour détecter tout débordement mémoire, mais elle permet très facilement de réduire les erreurs et cela gratuitement !

CHAPITRE 6

REGLES

La qualité logiciel est un élément crucial des applications. Le développeur passe plus de temps à localiser les erreurs qu'à rédiger le programme. En général, on retient le rapport 40% de développeur et 60% de débuggage. L'idéal serait de pouvoir écrire directement sans erreurs. Cela n'est malheureusement pas réaliste. Il faut chercher à réduire ce risque, sans pouvoir l'éliminer. A cet effet, on propose une liste de règles à suivre par les développeurs.

Tout langage est susceptible d'être bien ou mal employé. Le langage C++ peut accroître la qualité et l'expressivité des programmes. Un programmeur peu scrupuleux peut également rendre les choses opaques. La bonne programmation demande de la discipline et l'adhésion à des principes de qualité. Il y a une grande différence entre connaître et le faire correctement. Il n'est pas suffisant de connaître les principes fondamentaux du langage et de savoir les assembler, il faut être capable de produire des programmes lisibles, devant survivre au-delà du besoin immédiat. Les programmes objets doivent pouvoir être réutilisables, extensibles et compréhensibles. Un bon style de développement y contribuera fortement.

Ce chapitre regroupe un ensemble de règles à respecter pour développer *sans erreurs* avec ce langage. Celles-ci sont le résultat d'une longue expérience en C et C++. Les explications détaillées sont reportées plus loin dans ce livre afin de ne pas noyer les règles dans leurs descriptions. Si vous le désirez, vous pouvez sauter ce chapitre dans un premier temps pour

regarder précisément les descriptions des règles, puis vous reviendrez sur celui-ci par la suite. Vous pouvez utiliser ce chapitre comme « aide mémoire ». Si un point particulier vous intrigue, reportez-vous à la page indiquée afin d'y trouver des explications complémentaires.

Le « pourquoi » est plus important que le « quoi », mais il est plus facile de se souvenir d'un certain nombre de règles, que de mémoriser l'ensemble de ce livre. Les explications sont généralement un bon moyen pour les mémoriser. Ayant compris le « pourquoi », vous en déduirez vous-même le « quoi ». Par la suite, vous serez automatiquement interpellé par les risques d'erreurs de vos programmes.

Contrairement aux ouvrages courants traitant du C++, vous ne trouverez pas des chapitres traitant d'un concept particulier du langage. Par exemple, il n'existe pas de chapitre traitant exclusivement des `template` ou des références. C'est justement, lors du mélange de ces concepts que les difficultés arrivent.

Les règles sont regroupées selon l'objectif de leurs utilisations :

- applicables au C ou au C++,
- applicables au C seulement,
- applicables au C++ seulement,
- éviter les erreurs et les détecter,
- gestion de la mémoire,
- optimisation,
- portabilité.

Par ailleurs, d'autres règles ne concernent que le style de développement. Il est conseillé de les suivre pour faciliter la relecture de vos programmes, ou pour permettre de l'adapter à différents contextes.

Cette approche peut paraître déroutante. C'est pourquoi un index permet de remonter aux pages traitant d'un sujet particulier du langage. De plus, des renvois multiples parsèment cet ouvrage pour faciliter l'utilisation quotidienne de ce livre. Vous serez peut être amené à sauter de page en page pour résoudre un problème particulier. L'index vous y aidera. Un tableau récapitulatif des règles vous permettra de remonter aux règles concernées par une utilisation particulière du langage.

Il est à noter que certaines de ces règles sont contradictoires. En effet, on peut être amené à ne pas les respecter si cela est absolument nécessaire. Par exemple, il est indiqué qu'il ne faut pas utiliser d'objets globaux. Si vous êtes obligé de le faire, il est recommandé de les déclarer dans les fonctions mais il faudra alors faire en sorte que cela soit compensé par une information claire dans votre programme. Toute violation d'une règle doit pouvoir être

justifiée. Vous devez être capable d'expliquer techniquement pourquoi, vous avez été obligé de ne pas la suivre. Un commentaire peut d'ailleurs indiquer cette contrainte.

La partie à mon sens, essentielle, est celle intitulée « Règles de debug ». Les règles regroupées ici, vous éviteront des heures de travail passées à localiser les dysfonctionnements de vos programmes.

A. SOURCES C ET C++

a. C-CPP.DEB Règles de debug

C-CPP.DEB.1 Préférer les erreurs de compilation aux erreurs d'exécution.
Utilisez `#error` si nécessaire.

C-CPP.DEB.2 Renforcer les programmes avec des vérifications d'assertion.
Cela permet d'avoir une version de développement qui détecte les erreurs dès qu'elles arrivent (Voir « Invariant, Pré et Postconditions », page 238).

C-CPP.DEB.3 Construire des types pour tous les objets.
Ne pas utiliser `float` par exemple pour un prix.

C-CPP.DEB.4 Chaque variable doit prendre une valeur avant d'être utilisée.
Initialiser toutes les variables, même à zéro.

C-CPP.DEB.5 Allouer toujours le nombre exact d'octets d'un tampon.
Ne pas rajouter quelques octets pour *réduire* le risque d'erreur. Cela permet de rapidement détecter les erreurs de débordement de tampon.

C-CPP.DEB.6 Éviter au maximum les conversions de type.
Les conversions sont de la responsabilité du programmeur, pas du compilateur. Beaucoup d'erreurs viennent des conversions.

C-CPP.DEB.7 Déclarer `unsigned char` si vous utilisez l'Ascii 8 bits.
Le type `char` ne peut pas contenir l'Ascii 8 bits. Les accents français ne sont plus valides en Ascii 7 bits.

C-CPP.DEB.8 Ne pas convertir un type `const` en un type non-`const`.

Sauf exception, (voir la règle CPP.STY.6 page 222). L'attribut `const` permet de limiter les modifications d'un objet. Cela permet au développeur d'éviter les effets de bords. A la recherche d'une erreur, un développeur va considérer qu'une méthode recevant un pointeur `const`, ne va pas modifier l'objet. Il va négliger la consultation du corps de celle-ci. Si vous modifiez l'attribut, le développeur ne le saura pas lors de la lecture de la signature de la méthode.

C-CPP.DEB.9 Ouvrir les fichiers avec les droits de partage adéquats, et toujours vérifier l'ouverture d'un fichier.

Si un fichier ne doit être ouvert que par une seule application, il faut ouvrir le fichier avec les droits correspondants.

C-CPP.DEB.10 Chaque `switch` doit avoir une clause `default`.

Éventuellement avec `assert(0)`, si cette clause ne doit jamais être appelée.

C-CPP.DEB.11 Éviter les fonctions trop longues, type `switch` énormes.

Elles empêchent le compilateur d'optimiser et ne facilitent pas la relecture.

C-CPP.DEB.12 Tous les paramètres des macros doivent avoir des parenthèses dans celles-ci.

L'appel d'une macro avec un opérateur peut modifier son fonctionnement s'il n'y a pas de parenthèses.

C-CPP.DEB.13 Ne jamais utiliser une mémoire qui vient d'être libérée

Voir « `::NEW` de debug », page 130.

b. C-CPP.INC Includes

C-CPP.INC.1 Les classes qui ne sont utilisées qu'avec un pointeur ou une référence, ne doivent pas être incluses dans les fichiers d'en-tête (`*.h`).

Utiliser alors les déclarations *forward* (Voir « Comment optimiser la compilation », page 103).

C-CPP.INC.2 Ne jamais dupliquer une information dans plusieurs fichiers.

Utiliser les `.h` pour cela.

C-CPP.INC.3 Ne pas utiliser de nom absolu pour les fichiers `include`s.
Utiliser le paramètre du compilateur `-I` à la place.

C-CPP.INC.4 Configurer le compilateur pour faire apparaître le maximum de warnings.
Les warnings indiquent toujours une utilisation ambiguë du langage. Il ne faut pas les négliger.

C-CPP.INC.5 Rajouter tous les `include`s nécessaires à l'utilisation d'une fonction.
Il faut utiliser les prototypes pour chaque fonction.

C-CPP.INC.6 Tous les fichiers inclus doivent avoir un mécanisme pour prévenir les inclusions multiples.
Voir « Comment optimiser la compilation », page 103.

C-CPP.INC.7 Un fichier `include` doit se suffire à lui-même.
L'utilisateur d'un fichier ne doit pas connaître les autres fichiers dont il dépend.

C-CPP.INC.8 Utiliser `#include "..."` pour les includes de l'application, et `#include <...>` pour les includes des librairies.

c. C-CPP.MEM Règles de gestion mémoire

C-CPP.MEM.1 Toujours vérifier le retour d'un `malloc` ou d'un `new` avec `NULL`.
Éventuellement, dans un premier temps, ajouter un `assert(pt!=NULL)` juste après le `malloc` avant de gérer correctement l'erreur.

C-CPP.MEM.2 Libérer toute la mémoire utilisée et dans tous les cas, vérifier surtout les gestions d'erreurs.
Éventuellement, ajouter des compteurs de `malloc` et de `free`. Vérifier que tout est correct aux endroits clefs par un `assert`, ou bien, utiliser des librairies de debug pour `::new` et `::delete` (Voir « `::NEW` de debug », page 130).

d. C-CPP.OPT Règles d'optimisation

C-CPP.OPT.1 Ne pas utiliser `short` à la place de `int` si cela n'est pas utile (Page 168).

C-CPP.OPT.2 Utiliser `unsigned` si la variable n'a pas de raison d'être signée.

C-CPP.OPT.3 Utiliser « `"..." "..."` » pour la concaténation des chaînes de caractères constantes.

> Ne pas utiliser `"%s%s"` ou `<< "..." << "..."`. pour additionner deux chaînes constantes. `sprintf(buf, "%s%s", __FILE__, " erreur")` peut être judicieusement modifié en `strcpy(buf, __FILE__ " erreur")`.

e. C-CPP.POR Règles de portabilité

C-CPP.POR.1 Initialiser toutes les variables statiques et globales, au besoin à zéro.

> Cela facilite la portabilité sur d'autres OS ou environnements. On n'est jamais sûr qu'une variable est forcée à zéro malgré ce que demande la norme.

C-CPP.POR.2 Éviter au maximum d'utiliser les spécificités d'un microprocesseur ou d'une implantation de fichier.

> Si toutefois cela s'avérait nécessaire, encadrer le code par un « `#ifdef MSDOS` ». Par exemple, les noms de fichiers sous DOS sont toujours en majuscules, encadrer le `strupr()`.

C-CPP.POR.3 Ne pas assumer que les opérateurs d'une expression sont dans un ordre particulier.

> Les expressions ne sont pas exécutées dans le même ordre suivant les compilateurs. `(a++ + b)*(++a + c)` peut donner un résultat différent suivant le compilateur utilisé.

C-CPP.POR.4 Ne pas présumer de l'alignement d'une structure.

> Rien n'indique que deux attributs d'une structure sont accolés en mémoire.

C-CPP.POR.5 Ne pas assumer l'emplacement des types en mémoire.

Certains CPU demandent l'alignement de certains types sur des adresses particulières de la mémoire. Par exemple, le type `long` ne peut pas être sur une adresse impaire sur certaines machines.

C-CPP.POR.6 Ne pas sous-entendre l'ordre des octets dans un `int`, un `long` ou un pointeur. Utiliser une `union` à la place.

L'ordre des octets pour un type donné dépend du microprocesseur utilisé. Intel® utilise : poids faible, poids fort ; Motorola® utilise : poids fort, poids faible.

C-CPP.POR.7 Ne pas assumer que `char` est signé ou non signé. Toujours spécifier le type si cela est pertinent.

C-CPP.POR.8 Ne pas assumer que « `sizeof(int)==sizeof(long)` ».

Le type `int` est au minimum sur 16 bits, et le type `long` est au minimum sur 32 bits.

C-CPP.POR.9 Ne pas assumer que « `sizeof(void*)==sizeof(int)` ».

Si le type `int` est sur 16 bits, les adresses ne sont généralement pas de cette taille. Sinon, seuls 64 Ko seraient disponibles.

C-CPP.POR.10 Ne pas assumer que `int` est en 32 bits, mais plutôt en 16 bits.

C-CPP.POR.11 Ne pas considérer la taille d'un type de données comme nombre de bits.

Utiliser des « `typedef unsigned char bit8` ». Vous pouvez aussi utiliser les champs de bits en structure. Il existe des machines avec le type `char` en 9 ou 10 bits !

C-CPP.POR.12 Ne pas écrire « `if (test)` » ou « `if (!test)` » pour un pointeur mais comparer avec la valeur `NULL`.

Voir page 208, règle CPP.DEB.39.

C-CPP.POR.13 Ne pas utiliser d'identifiants globaux commençant par un ou plusieurs soulignés (`_x` ou `__x`).

Ceux-ci sont réservés pour les compilateurs. Les attributs d'une classe peuvent commencer par un souligné car ils sont préfixés par le nom de la classe.

C-CPP.POR.14 N'utiliser les opérateurs de décalage binaire que sur des types non signés.

Le résultat d'un décalage binaire sur des types signés est indéfini. Cela dépend du compilateur.

C-CPP.POR.15 Ne pas considérer que « `*(char*)NULL=='\0'` ».

Cela dépend des compilateurs.

f. C-CPP.TYP Règles de typage

C-CPP.TYP.1 Utiliser au maximum l'attribut `static` pour une fonction si elle n'est utilisée que dans un seul fichier.

Cela permet de ne pas y attacher trop d'importance lors de la relecture du source et évite les conflits de noms lorsque l'on développe à plusieurs.

C-CPP.TYP.2 Utiliser au minimum les variables globales dans un source, afin d'éviter les effets de bord lors de l'appel de fonction.

Ceci facilite aussi le portage en multi-thread. Si une fonction doit utiliser une variable globale uniquement pour elle, il faut la déclarer en `static` dans la fonction. Si une variable globale n'est utilisée que dans un source, l'indiquer également en `static`.

C-CPP.TYP.3 Utiliser l'attribut `const` si un pointeur n'est utilisé qu'en lecture.

Cela permet en relisant le source d'être rapidement sûr de la non influence d'une fonction sur un objet pointé.

g. C-CPP.STY Règles de style

C-CPP.STY.1 Documenter le plus possible.

Documenter les routines avec les paramètres en entrée et en sortie. Indiquer si les pointeurs reçus sont adoptés par la routine ou si l'appelant peu modifier l'objet pointé par la suite. Dans le code, indiquer par une ligne, les étapes de la routine. Traduire les lignes de code, c'est-à-dire, indiquer le but de certaines d'entre elles.

C-CPP.STY.2 Rédiger une documentation parallèlement à votre développement afin de transmettre l'explication de la philosophie du programme ainsi que les choix techniques.

Ce fichier peut également servir à noter : les points à éclaircir sur le programme ; les choses à faire ainsi que les évolutions futures à envisager.

C-CPP.STY.3 Chaque variable doit être déclarée séparément.
Cela permet de documenter chacune d'elles.

C-CPP.STY.4 Choisir les noms de variables suivant leurs usages.
Le nom des variables doit être immédiatement compréhensible, éviter les abréviations.

C-CPP.STY.5 Une variable avec un scope long, doit avoir un nom long.
Une variable globale ne doit pas être cachée par une variable locale. Les noms longs réduisent ce risque.

C-CPP.STY.6 Les variables doivent être déclarées avec le scope le plus petit.
Plus une variable est détruite tôt, moins elle peut interférer avec l'environnement.

C-CPP.STY.7 Utiliser `typedef` pour simplifier les déclarations des pointeurs de fonctions.
La syntaxe de déclaration de pointeur de fonction est illisible. Le `typedef` permet de simplifier grandement l'écriture.

C-CPP.STY.8 Envisager toujours la possibilité de réutiliser les routines.
Pour cela, bien les découper en couches et les documenter.

C-CPP.STY.9 Ne pas hésiter à découper une fonction en plusieurs plus petites, même si les sous-fonctions ne sont appelées qu'une fois.
Cela permet une relecture plus facile et le compilateur est là pour remettre les fonctions dans l'appelant.

C-CPP.STY.10 Séparer les fonctions de traitement, des fonctions d'interface utilisateur.
Même si cela entraîne l'appel en cascade de fonctions. Cela permet de porter le programme sur différents OS. Le linker fera le raccourci des appels.

C-CPP.STY.11 N'optimiser le code que s'il y a un problème de performance.

Pour optimiser correctement, il est préférable d'avoir une version lente mais simple, pour pouvoir comparer les résultats de la version optimisée. Il est alors utile d'écrire dans un premier temps la version simple. Une analyse par la suite pourra détecter les routines véritablement critiques devant être modifiées. Il ne faut pas consacrer du temps à optimiser un traitement ne représentant que 1% du temps de l'application.

C-CPP.STY.12 Éviter toute *verrue*.

Ne pas hésiter à modifier même en profondeur le programme. C'est, paradoxalement, du temps gagné car ce programme sera plus facilement modifiable et transférable à un autre développeur.

C-CPP.STY.13 Pour le développement à plusieurs, ne pas prendre d'initiatives sur les liaisons entre les couches sans en discuter avec tout le monde.

Tout est possible au sein de sa propre tâche mais pas entre les tâches. Ne jamais sous-entendre qu'une autre couche est écrite d'une certaine façon. En règle générale, éviter tout sous-entendu. Ce qui n'est pas écrit clairement n'est pas à prendre en compte. Tout peut changer pour régler des contraintes techniques.

C-CPP.STY.14 Privilégier la relecture d'un programme à sa rapidité.

Un programme doit pouvoir être modifié par d'autres. Il est difficile, même pour l'auteur de reprendre un ancien programme.

C-CPP.STY.15 Toujours indiquer le type de retour d'une fonction.

Il n'est pas nécessaire d'indiquer le type de retour d'une fonction. Par défaut, le compilateur utilise le type `int`. Il ne faut pas utiliser cette syntaxe.

C-CPP.STY.16 Ne pas utiliser de `goto`.

Étonnant non ?

C-CPP.STY.17 Utiliser `for(;;)` à la place de `while(1)`.

La sémantique des deux écritures est la même, mais l'existence de la constante `1` n'est pas justifiée. La syntaxe `for` est là pour cela.

C-CPP.STY.18 Utiliser `continue` ou `break` pour modifier le cycle d'une boucle.

Ces deux mots clefs permettent de modifier le cycle d'une boucle. Ils évitent d'utiliser un `goto`.

C-CPP.STY.19 Utiliser les parenthèses pour éclairer l'ordre d'évaluation des expressions.

> Les parenthèses ajoutent une sémantique aux expressions. Elles permettent de comprendre les étapes ayant guidé la rédaction de celle-ci.

B. SOURCE C

a. C.POR Règles de portabilité

C.POR.1 Ne pas limiter un programme à un modèle mémoire donné.

> Toujours envisager tous les modèles en même temps. Windows 16 bits est particulièrement exigeant sur les modèles mémoire.

b. C.STY Règles de style

C.STY.1 Toutes les fonctions doivent avoir un prototype.

C.STY.2 Éviter au maximum les constantes numériques.

> Utiliser des `#define` à la place. Cela permet de modifier facilement le programme.

C.STY.3 Utiliser des mots tout en majuscules pour les `#define`.

C.STY.4 Utiliser `typedef` plutôt que `struct x`.

C.STY.5 Déclarer en `extern` les objets pouvant être utilisés par un autre fichier.

> Les `extern` doivent être déclarés dans les fichiers `.h`.

C.STY.6 Pour les librairies avec plusieurs fonctions cohérentes entre elles, utiliser une approche « *objet* ».

Cela veut dire, déclarer un *constructeur* pour initialiser l'utilisation de la librairie, un *destructeur* pour sa fermeture, et regrouper les fonctions. Même s'il n'est pas nécessaire d'avoir un constructeur, le simuler avec un `#define`. Cela permet de faciliter la modification future du programme.

C. SOURCE C++

a. CPP.DEB Règles de debug

CPP.DEB.1 Toujours déclarer, pour tout objet ayant un pointeur, le constructeur de copie et l'opérateur d'affectation.
Si ces fonctions ne doivent jamais être appelées, les déclarer en `private`, sinon les écrire correctement (Page 169).

CPP.DEB.2 Toujours utiliser `delete []` pour les objets alloués avec un `new []` (Page 172).

CPP.DEB.3 Toujours effacer les tableaux de types standard par `delete []()` (Page 174).

CPP.DEB.4 Toujours déclarer le destructeur en `virtual` pour les classes ayant une méthode virtuelle (Page 175).

CPP.DEB.5 Toujours utiliser les pointeurs de membres pour accéder dynamiquement à une méthode d'un objet (Page 176).

CPP.DEB.6 Toujours utiliser des références pour les objets ayant des méthodes virtuelles (Page 178).

CPP.DEB.7 Toujours ajouter, pour tout `operator =()`, au début de la méthode : « `if (this==&x) return *this;` » ou « `assert(this!=&x);` » (Page 180).

CPP.DEB.8 Toujours rédiger un `template` en pensant à un objet paramétrable du type référence (Page 181).

CPP.DEB.9 Toujours envisager les exceptions lors de l'écriture d'un constructeur.
Voir « Comment gérer les erreurs dans les constructeurs », page 83.

CPP.DEB.10 Toujours prévoir le passage d'un `throw` dans un objet.
Un `new` doit pouvoir exécuter un `throw` (Voir « Exceptions », page 289) !

CPP.DEB.11 Toujours initialiser les éléments dans l'ordre de déclaration dans l'objet (Page 182).

CPP.DEB.12 Toujours utiliser les pointeurs de membres plutôt que l'offset d'un attribut (Page 183).

CPP.DEB.13 Toujours écrire un `operator <<()` pour tous les objets afin de faciliter les traces.
Voir « Pourquoi utiliser <iostream.h> », page 119.

CPP.DEB.14 Ne jamais appeler de méthodes virtuelles dans un constructeur ou un destructeur (Page 184).

CPP.DEB.15 Ne jamais renvoyer une référence sur une variable locale ou un paramètre (Page 186).

CPP.DEB.16 Ne jamais déclarer de destructeur en `virtual` pur (Page 187).

CPP.DEB.17 Ne jamais mélanger les allocations *via* `::malloc` et celle *via* `::new` (Page 188).

CPP.DEB.18 Ne jamais utiliser de surcharge dans un `template` avec le même nombre de paramètres pour les déclarations de méthodes (Page 189).

CPP.DEB.19 Ne jamais utiliser la fonction `exit()` en C++ (Page 189).

CPP.DEB.20 Ne jamais utiliser les macros `va_start`, `va_arg` et `va_end` en C++ (Page 191).

CPP.DEB.21 Ne jamais utiliser `longjmp` en C++ (Page 191).

CPP.DEB.22 Ne jamais utiliser `qsort` avec des objets C++ (Page 193).

CPP.DEB.23 Ne jamais appeler `throw` avec un objet statique.
Voir « Exceptions », page 289.

CPP.DEB.24 Ne jamais utiliser `throw` avec un pointeur, sauf s'il s'agit de l'adresse d'un objet statique.
Voir « Exceptions », page 289.

CPP.DEB.25 Éviter d'utiliser des références dans les classes (Page 194).

CPP.DEB.26 Éviter les interdépendances entre objets globaux (Page 195).

CPP.DEB.27 Éviter d'utiliser des objets temporaires pour initialiser les objets globaux (Page 200).

CPP.DEB.28 Ne pas déclarer de variables après un label de `switch`.
La visibilité de ces variables est violée par les `case`s successifs.

CPP.DEB.29 La responsabilité d'un destructeur est de détruire les ressources allouées lors de la vie de l'objet.
et pas uniquement lors de la construction de celui-ci.

CPP.DEB.30 Déclarer les opérateurs par groupe pour garder une cohérence sémantique.
Déclarer `operator ==()` avec `operator !=()` (Voir « Écritures génériques », page 99).

CPP.DEB.31 Utiliser les parenthèses pour encadrer les opérations lors des manipulations de flux (Page 202).

CPP.DEB.32 Toute conversion de référence est suspecte (Page 202).

CPP.DEB.33 Un objet temporaire est détruit à la fin de l'expression (Page 203).

CPP.DEB.34 Si une méthode virtuelle pure est appelée à l'exécution, cela provient certainement de l'appel d'une méthode virtuelle dans un constructeur (Page 205).

CPP.DEB.35 Pour rédiger un cache en surchargeant l'`operator ->()` utilisez un LRU (Last Recent Use) (Page 205).

CPP.DEB.36 Lors d'un appel de fonction avec un nombre variable de paramètres recevant la valeur `NULL`, convertir celle-ci en `(void*)`.
Voir « Différences entre le C et le C++ », page 307.

CPP.DEB.37 Ne jamais déclarer de variable statique dans une fonction `inline`.
Celle-ci serait présente dans chaque module objet et utilisée individuellement dans chacun des modules.

CPP.DEB.38 Toujours utiliser l'opérateur de scope le plus proche possible dans la hiérarchie de classes.
Voir « Utilisation générique de la classe héritée », page 32.

CPP.DEB.39 Ne pas surcharger une fonction ou une méthode avec un type numérique et un pointeur (Page 208).

CPP.DEB.40 Éviter les écritures ambiguës lors de la rédaction des conversions et des constructeurs (Page 209).

CPP.DEB.41 Ajouter toujours la syntaxe classique de l'opérateur `new` lors de la surcharge de celui-ci (Page 210).

CPP.DEB.42 Ne pas déclarer des classes « en avant » si elles héritent l'une de l'autre (Page 211).

CPP.DEB.43 Ne pas renvoyer de pointeur ou de référence non `const` sur un attribut d'un objet dans une méthode non `const`.
Certains compilateurs signalent ce problème (Voir « Quand et où utiliser les références », page 111).

CPP.DEB.44 Éviter de redéfinir une méthode non virtuelle lors d'un héritage.
Le polymorphisme n'étant pas pris en compte par cette écriture, l'utilisateur risque d'être perturbé.

CPP.DEB.45 Ne pas modifier les paramètres par défaut lors de la surcharge d'une méthode virtuelle.
La sémantique et l'interface des méthodes virtuelles doivent être uniques.

CPP.DEB.46 Maîtriser quelles méthodes le C++ écrit et appelle en silence.

b. CPP.OPT Règles d'optimisation

CPP.OPT.1 Toujours initialiser les éléments d'un objet en dehors des accolades du constructeur (Page 212).

CPP.OPT.2 Toute utilisation d'un opérateur suffixe doit être justifiée par l'utilisation de l'objet avant le traitement de l'opérateur.
Utiliser `++i` et non `i++` (Page 213).

CPP.OPT.3 Passer, par principe, les paramètres d'une fonction en référence constante.
Éviter le passage par valeur (Voir page 178, règle CPP.DEB.6).

CPP.OPT.4 Renvoyer directement le retour d'une fonction pour éviter la création d'objets temporaires (Page 214).

CPP.OPT.5 Toujours préférer l'initialisation à la place de l'assignation (Page 215).

CPP.OPT.6 Envoyer (`throw`) les exceptions par valeur et les capturer (`catch`) par référence (Page 215).

CPP.OPT.7 Une fonction ne doit pas exécuter un code fondé sur la valeur d'un argument (défaut ou non) s'il n'est pas valorisé à l'exécution.
Dans ce cas, déclarer deux ou plusieurs méthodes (Page 216).

CPP.OPT.8 Utiliser `operator +=()` pour écrire `operator +()` (Page 217).

c. CPP.POR Règles de portabilité

CPP.POR.1 Ne pas considérer le type `char` comme un type `int` (Page 219).

CPP.POR.2 Si une variable doit être utilisée après une boucle, la déclarer en dehors de celle-ci (Page 221).

CPP.POR.3 Évitez d'utiliser un paramètre reçu par référence dans un constructeur (Page 221).

CPP.POR.4 Encadrer les variables globales, les constantes, les types énumérés et les `typedef` dans une classe.
Cela permet une compatibilité avec les futurs `namespace` (Voir « Prévoir le `NAMESPACE` », page 41).

CPP.POR.5 Si une méthode reçoit un « `const char*` » comme paramètre, rédiger des méthodes surchargées avec : « `const signed char*` » ; « `const unsigned* char` », et `string` comme paramètre.
Voir « Recevoir un paramètre `CONST CHAR*` », page 36.

d. CPP.STY Règles de style

CPP.STY.1 Relire vos classes pour être sûr que les relations *est-un, a-un* et *utilise-un* ont un sens.

CPP.STY.2 L'utilisateur ne doit pas connaître ce que le créateur a fait pour rédiger la classe.
Seul l'interface doit être public. Cela permet de modifier les méthodes en profondeur en respectant l'interface.

CPP.STY.3 Cacher au maximum les attributs d'une classe.
Cela permet au créateur de la classe de pouvoir modifier profondément celle-ci sans perturber les développements en cours (Voir « L'accès aux attributs », page 42).

CPP.STY.4 Simplifier au maximum l'utilisation d'une classe.
Utiliser des attributs par défaut, plusieurs constructeurs, ...

CPP.STY.5 Un opérateur d'assignation `operator =()` doit toujours renvoyer une référence sur l'objet lui-même.
Cela permet une écriture du type `a=b=c;`.

CPP.STY.6 Toujours ajouter l'attribut `const` à tous les niveaux, dans les paramètres, les retours de fonctions, les références, et le type des méthodes (Page 222).

CPP.STY.7 Utiliser `friend` pour obtenir de l'extérieur, les droits sur un objet plutôt que rendre `public` les éléments (Page 228).

CPP.STY.8 Ne pas convertir en type de base, un pointeur d'un objet hérité de façon `protected` (Page 229).

CPP.STY.9 Éviter de convertir un objet en `void*` (Page 230).

CPP.STY.10 Ne pas déclarer de référence artificielle sur l'adresse `NULL` (Page 231).

CPP.STY.11 Toujours utiliser une référence constante si l'objet référencé n'est pas modifié (Page 231).

CPP.STY.12 Déclarer si c'est possible, les objets statiques dans la fonction qui les utilise plutôt qu'en dehors de celle-ci (Page 232).

CPP.STY.13 Déclarer en `extern` les objets pouvant être utilisés par un autre fichier (Page 233).

CPP.STY.14 Déclarer dans le prototype d'une fonction les crochets d'un paramètre tableau si la fonction ou la méthode utilise le pointeur comme un tableau (Page 234).

CPP.STY.15 Ne pas utiliser d'héritage privé.
> Utiliser une association privée à la place.

CPP.STY.16 Rédiger les classes les plus atomiques possibles.
> Chaque classe doit être simple et avoir un rôle clair. Si une classe devient trop compliquée, faites en plusieurs simples. Vous en casserez la complexité.

CPP.STY.17 Lors de la déclaration de méthodes surchargées, toutes doivent avoir la même sémantique.

CPP.STY.18 Minimiser l'utilisation des variables temporaires.

CPP.STY.19 Ne pas déclarer de macro avec `#define`, mais utiliser `inline`.
> Les fonctions `inline` évitent les soucis des macros et sont plus puissantes.

CPP.STY.20 Ne pas réécrire tout le code C en C++ si vous n'avez pas de changements majeurs à y effectuer.

> Les programmes C sont parfaitement interfaçables avec les programmes C++.

CPP.STY.21 Utiliser `const` ou `enum` à la place de `#define`.

> Le `#define` ne possède pas de type.

CPP.STY.22 Utiliser le type de classe le moins riche dans la hiérarchie suivant l'utilisation qui en est faite (Page 235).

CPP.STY.23 Utiliser les flux du C++ à la place de `stdio.h`.

> Voir « Pourquoi utiliser <iostream.h> », page 119.

CPP.STY.24 Utiliser `true`, `false` et le type `bool`. Ne pas utiliser `++b` pour mettre à `true`.

> Voir « Comment implanter le type `BOOL` », page 34.

CPP.STY.25 Toujours utiliser l'opérateur de résolution (`::`) dans une méthode pour appeler une fonction.

> Si plus tard, la classe de base ajoutait une méthode synonyme à une fonction déjà utilisée, vous risqueriez d'appliquer cette méthode lors de la recompilation. Cela ne conviendrait pas.

CPP.STY.26 Ne pas maintenir la liste des instances d'une classe dans les classes elles-mêmes.

> Utiliser un objet *contenant* distinct pour cela.

CPP.STY.27 Éviter les `switch` dans les fonctions.

> Utiliser le polymorphisme à la place.

CPP.STY.28 Déclarer tous les types, propres à une classe, dans celle-ci.

> Déclarer les `enum`s, d'autres classes, ou les `typedef` dans la classe.

CPP.STY.29 Si un constructeur reçoit une référence, indiquer en commentaire s'il utilise les valeurs de l'objet référencé.

> Si le constructeur adopte la référence livrée en paramètre, il faut faire attention à la durée de vie de l'objet référencé.

CPP.STY.30 Doubler les méthodes par des opérateurs si la sémantique est identique.
Exemple : `add()` et `operator +=()`.

CPP.STY.31 Envisager toujours la possibilité de réutiliser l'objet.
C'est un des objectifs de l'approche objet.

CPP.STY.32 Ne pas allouer un attribut dans le tas si celui-ci peut être présent dans l'objet lui-même.
Le tas n'est pas toujours disponible. Dans ce cas, l'objet est partiellement construit. Si l'attribut est présent dans l'objet lui-même, tout l'objet est dans le même contexte mémoire, la pile, globale, ou dans le tas.

CPP.STY.33 Centraliser les conversions identiques dans une seule méthode.
Ajouter une méthode, `private` si nécessaire, n'effectuant que la conversion, et l'utiliser dans les autres méthodes (Voir « Relation virtuelle », page 53).

CPP.STY.34 Lors de la redéfinissions d'une méthode virtuelle, appeler la méthode de la classe directement dérivée, même si elle n'est disponible que par un héritage supplémentaire.
Voir « Utilisation générique de la classe héritée », page 32.

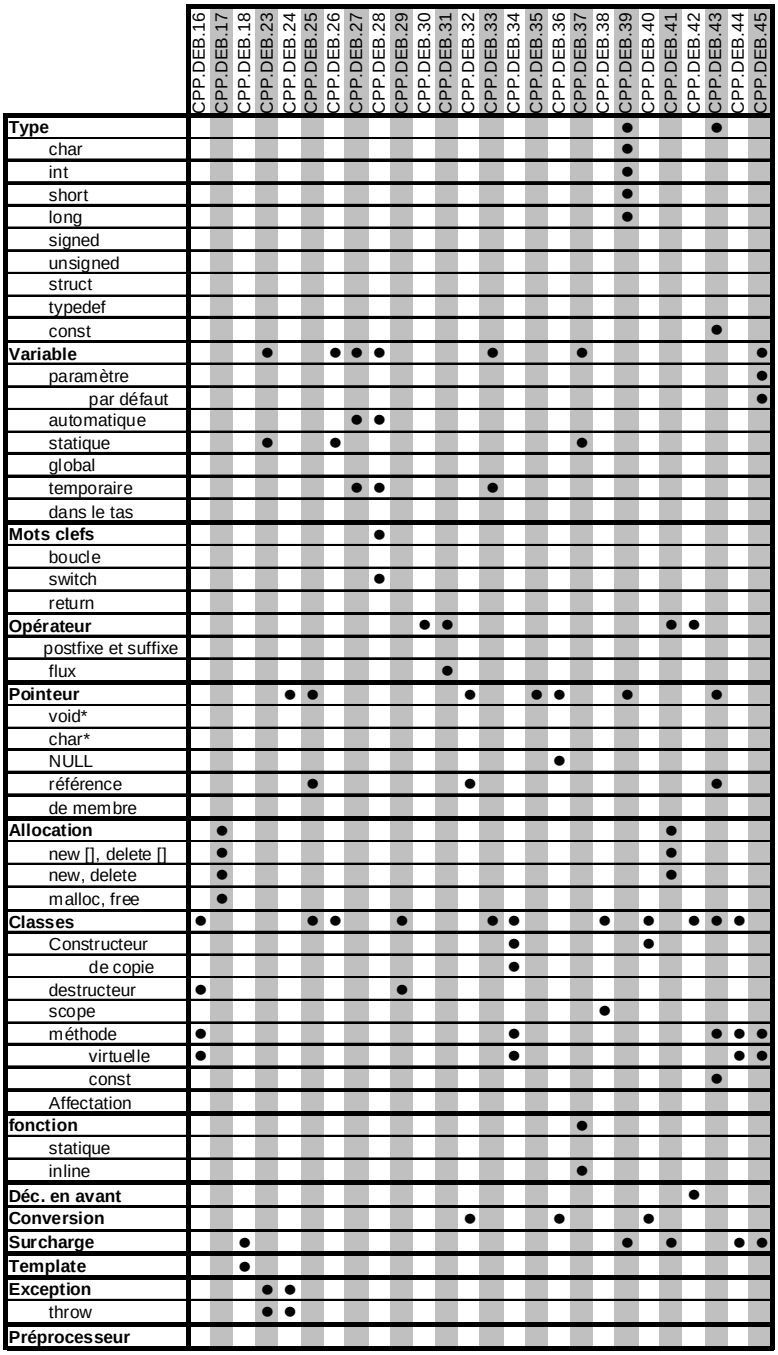
D. RECAPITULATION DES REGLES

Pour pouvoir facilement retrouver les règles concernées par un ensemble de plusieurs services offerts par le C++, voici un tableau récapitulatif de celles-ci avec les concepts C++ concernés.

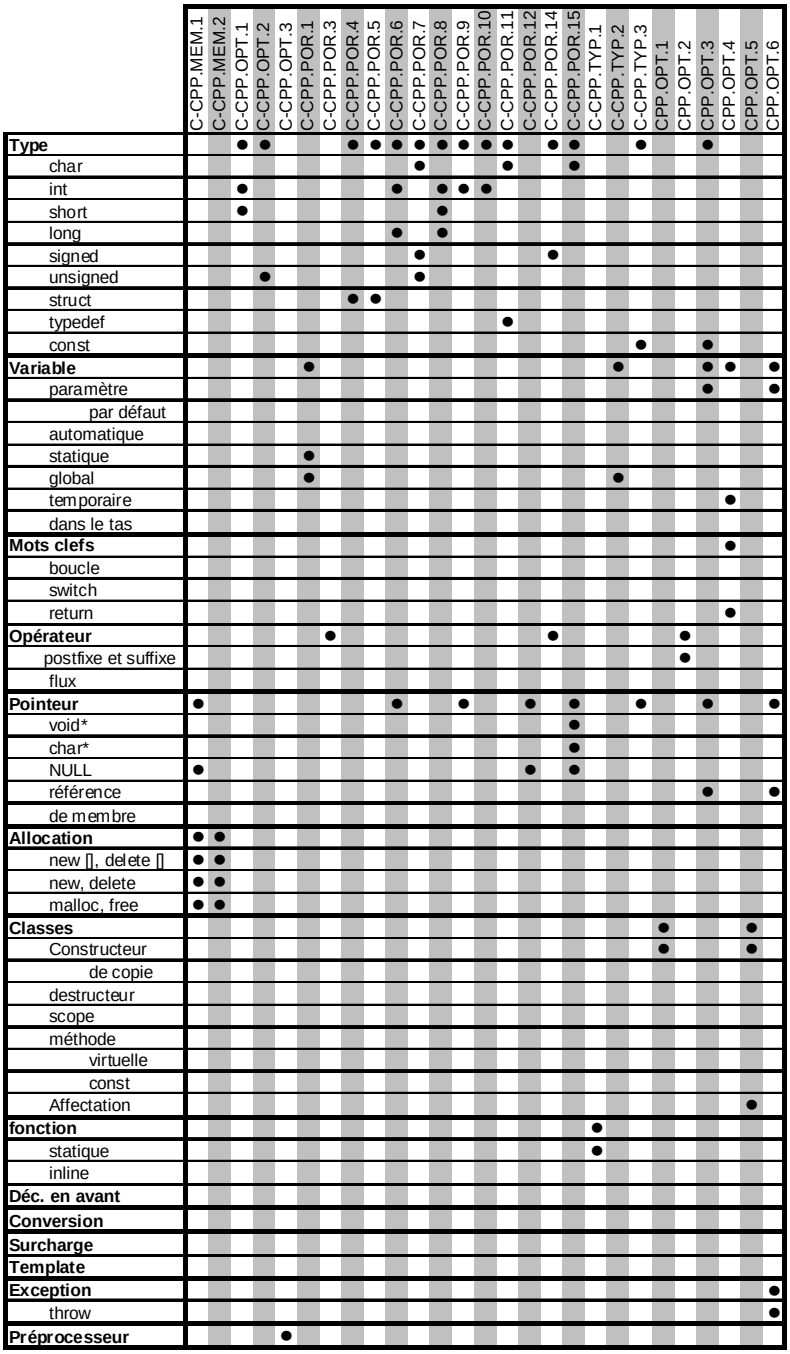
Ce tableau regroupe les règles de debug. Lors de la détection d'une erreur, consultez-le et localisez les règles concernant une utilisation particulière du langage.

	C-CPP.DEB.3	C-CPP.DEB.4	C-CPP.DEB.5	C-CPP.DEB.6	C-CPP.DEB.7	C-CPP.DEB.8	C-CPP.DEB.10	C-CPP.DEB.11	C-CPP.DEB.12	C-CPP.DEB.13	CPP.DEB.1	CPP.DEB.2	CPP.DEB.3	CPP.DEB.4	CPP.DEB.5	CPP.DEB.6	CPP.DEB.7	CPP.DEB.8	CPP.DEB.9	CPP.DEB.10	CPP.DEB.11	CPP.DEB.12	CPP.DEB.13	CPP.DEB.14
Type	●				●	●																		
char					●																			
int																								
short																								
long																								
signed																								
unsigned					●																			
struct																								
typedef	●																							
const						●																		
Variable		●																						
paramètre																								
par défaut																								
automatique		●																						
statique		●																						
global		●																						
temporaire																								
dans le tas																								
Mots clefs							●	●																
boucle																								
switch							●	●																
return																								
Opérateur																	●						●	
postfixe et suffixe																								
flux																							●	
Pointeur										●	●	●	●		●	●		●				●		
void*																								
char*													●											
NULL																								
référence																●		●						
de membre															●							●		
Allocation			●							●		●	●											
new [], delete []			●									●	●											
new, delete			●																					
malloc, free			●																					
Classes											●			●	●	●			●		●			●
Constructeur											●								●		●			●
de copie											●								●					
destructeur														●										
scope																								
méthode														●	●	●								●
virtuelle														●		●								●
const																								
Affectation											●													
fonction																								
statique																								
inline																								
Déc. en avant																								
Conversion				●		●																		
Surcharge																								
Template																		●						
Exception																			●	●				
throw																								
Préprocesseur									●															

La suite du tableau de debug.



Le tableau suivant reprend les autres règles, celles concernant l'optimisation, le style de développement ou autres.



La suite du tableau.

	CPP.POR.1	CPP.POR.2	CPP.POR.3	CPP.POR.4	CPP.POR.5	C-CPP.TYP.1	C-CPP.TYP.2	C-CPP.TYP.3	CPP.OPT.1	CPP.OPT.2	CPP.OPT.3	CPP.OPT.4	CPP.OPT.5	CPP.OPT.6	CPP.POR.1	CPP.POR.2	CPP.POR.3	CPP.POR.4	CPP.POR.5
Type	●			●				●			●				●			●	
char	●														●				
int	●														●				
short																			
long																			
signed																			
unsigned																			
struct																			
typedef				●														●	
const				●				●			●							●	
Variable		●	●	●			●				●	●		●		●	●	●	
paramètre			●								●			●			●		
par défaut																			
automatique		●														●			
statique				●														●	
global							●												
temporaire												●							
dans le tas																			
Mots clefs		●										●				●			
boucle		●														●			
switch																			
return												●							
Opérateur										●									
postfixe et suffixe										●									
flux																			
Pointeur			●	●				●			●			●			●		●
void*																			
char*					●														●
NULL																			
référence			●								●			●			●		
de membre																			
Allocation																			
new [], delete []																			
new, delete																			
malloc, free																			
Classes			●	●					●				●				●	●	
Constructeur									●				●						
de copie																			
destructeur																			
scope																			
méthode																			
virtuelle																			
const																			
Affectation													●						
fonction						●													
statique						●													
inline																			
Déc. en avant																			
Conversion																			
Surcharge																			
Template																			
Exception														●					
throw														●					
Préprocesseur																			

CHAPITRE 7

DESCRIPTION DES REGLES

Ce chapitre reprend en détails certaines règles déclarées dans le chapitre précédent. L'accent est porté sur les effets de bord du langage à maîtriser parfaitement.

Un petit exemple vous est proposé et vous devez trouver l'erreur insérée. L'explication vous est alors donnée juste en dessous, avec éventuellement, différentes techniques pour la contourner.

Chaque exemple permet d'introduire une règle de développement afin d'éviter que l'erreur ne se représente. Les règles de qualité se déduisent de l'expérience.

L'objectif est de faire connaître les sources d'erreurs les plus subtiles qui risquent d'exister dans tous les développements C++. Certains problèmes sont faciles, d'autres beaucoup moins. Tout programmeur C++ devrait être capable de répondre correctement à la majorité des problèmes exposés. Lors des développements, il faut bien prendre conscience des risques évoqués. Ceci permet d'écrire un programme sans *trop* d'erreurs, et surtout, de ne pas perdre de temps dans la localisation de celles-ci.

Les exercices suivants sont réduits à leur plus simple expression. Aucun `#include` n'est indiqué pour ne pas alourdir les sources. Toutes les allocations « mémoire » sont considérées réussies. Il n'existe pas de réponse du type : « L'allocation n'est pas vérifiée, il est donc possible d'écrire à l'adresse `NULL` ». Tous les problèmes doivent être portables et ne

pas dépendre d'une version particulière du compilateur. Certains d'entre eux fonctionnent avec un compilateur mais pas avec un autre. Il faut dans ce cas que vous indiquiez pourquoi.

Règle C-CPP.OPT.1

```
short add(short i, short j)
{ return i+j; }
```

Le compilateur peut annoncer un `warning`, pourquoi ?

Si « `sizeof(short)==sizeof(int)` », il n'y a pas de problème, le compilateur ne signale pas d'erreur. Par contre, si « `sizeof(short)<sizeof(int)` », un message est affiché. En effet, toutes les opérations sont faites en `int`. Les variables `i` et `j` sont dans un premier temps converties en entiers. L'addition est ensuite calculée, le résultat de celle-ci est un entier. Ensuite le compilateur cherche à convertir cet entier en `short`.

```
{ return (short)((int)i+(int)j) }
```

Il y a perte de précision. La donnée résultante peut être erronée, ce qui est signalé par un `warning`. Cela indique qu'il est inutile d'utiliser le type `short` pour optimiser un programme, mais que celui-ci n'est utile que pour réduire la taille mémoire du logiciel. Le compilateur passe beaucoup de temps à convertir les `short` en `int` et inversement, ce qui ralentit le programme. L'arithmétique `short` n'est pas présente dans les compilateurs. On peut le regretter.

De plus, le type `short` peut entraîner un problème de portabilité lors de la résolution des méthodes surchargées.

```
void f(int);          // f1
void f(unsigned int); // f2

unsigned short s;
f(s);
```

Si les types `short` et `int` sont de tailles identiques, l'appel de `f(s)` sera résolu sur l'appel de la version `f2`. Sinon, le type `int` est capable d'avoir toutes les valeurs possibles de `unsigned short`, alors la promotion de type, convertie le `unsigned short` en `int`, et c'est l'appel de `f1` qui est généré.

Ne pas utiliser `short` à la place de `int` si cela n'est pas utile.

Règle CPP.DEB.1

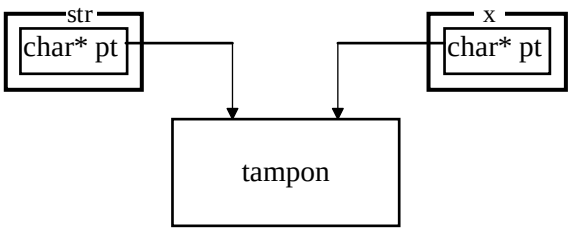

```
class CString
{ char* pt;
  public:
    CString(const char* src)
  { pt=new char[strlen(src)+1];
    strcpy(pt,src);
  }
    ~CString()
  { delete [] pt;
  }
};

CString f()
{ CString str("test");
  return str;
}

void main()
{ CString x=f();
}
```

C'est incorrect, pourquoi ?

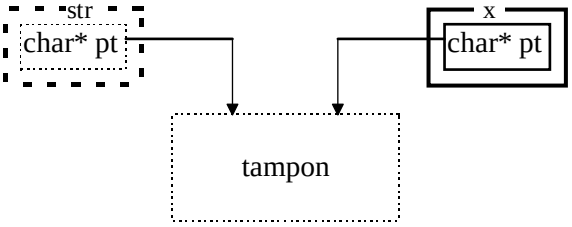
Pour bien comprendre l'erreur, il faut tout d'abord savoir comment le compilateur permet à une fonction de renvoyer un objet. La fonction `f()` reçoit un pointeur caché indiquant où l'objet retourné doit être créé. L'appel de la fonction `f()` se traduit comme cela :

```
void f(CString* _ret)                    // ret est un pointeur caché
{ CString _tmp("test");
  _ret->CString::CString(_tmp);          // Appel du ctr de copie
  _tmp::~CString();                      // Destruction de l'obj _tmp
}

void main()
{ CString x;                             // Déclaration de l'objet
                                         // sans construction
  f(&x);                                 // Appel de la fonction
}
```

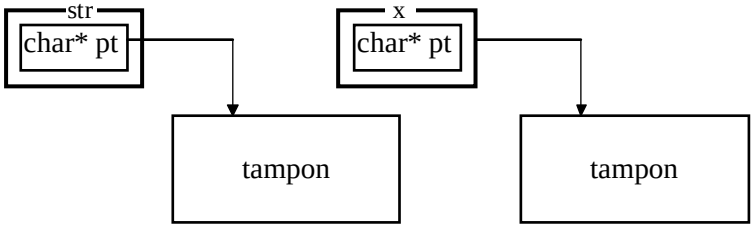
L'instruction `return` est traduite par l'appel du constructeur de copie sur le pointeur caché. La classe `CString` n'en possède pas. Le compilateur utilise alors le constructeur de copie par défaut. Celui-ci effectue une copie binaire membre à membre des éléments de la structure.



Le pointeur `pt` est recopié mais pas le tampon pointé par `pt`. Ensuite, le programme détruit l'objet `str`. Le tampon est aussi détruit.



L'objet copié pour le retour est erroné car le pointeur `pt` pointe sur une zone libérée ! Le constructeur de copie aurait dû donner :



Le plus grave est que cela passe généralement inaperçu car les données du tampon ne sont pas physiquement détruites. Si une version de debug de `::delete` remplit les zones à effacer avec une valeur quelconque avant de libérer la mémoire, cette erreur apparaîtra lors de l'utilisation du tampon. Vous pouvez vous en convaincre en ajoutant avant le « `delete [] pt` », la commande « `*pt='\0';` », afin de le détruire.

Cette erreur se présente aussi dans un autre cas.

```
void main()
{ CString a="abc";
  { CString b="def";
    a=b;                                    // Copie des pointeurs !
  }
  // Erreur ici
}
```

Cette fois, c'est l'opérateur d'affectation par défaut qui la génère. Lors de la destruction de l'objet b, a devient erroné.

Dans le cas présent il faut ajouter :

```
  private:
    CString(const CString& x);              // Non implementé
    CString& operator =(const CString& x);  // Non implementé
```

pour que le compilateur signale le problème. Une solution élégante pour régler ce cas de figure est d'utiliser un pointeur d'agrégation comme attribut de la classe (Voir « Durée de vie des objets » page 57).

```
class CString
{ CPtrArrayAggr<char> pt;
  public:
    CString(const char* src)
  : pt(new char[strlen(src)+1])
  { strcpy(pt,src);
  }
};
```

Avec cet outil, l'affectation, le constructeur de copie et le destructeur sont correctement générés par le compilateur.

Toujours déclarer, pour tout objet ayant un pointeur, le constructeur de copie et l'opérateur d'affectation.

Règle CPP.DEB.2

CString

```
class CString
{ char* pt;
  public:
    CString()
  { pt=(char*)::malloc(4);
    ::strcpy(pt,"ABC");
    cout << "CString::CString()" << endl;
  }
    ~CString()
  { ::free(pt);
    cout << "CString::~CString()" << endl;
  }
};

void main()
{ CString* pt=new CString[3];
  // ...
  delete pt;
}
```

Qu'affiche `main`, pourquoi et comment corriger ?

`main` affiche :

```
CString::CString()
CString::CString()
CString::CString()
CString::~CString()
```

Il n'y a plus de parité entre les constructeurs et les destructeurs !

Lors de la création dynamique d'un tableau d'objet, le compilateur alloue la mémoire nécessaire, puis appelle le constructeur vide pour chaque élément du tableau. Lors de la destruction de celui-ci, il ne fait pas la différence entre un pointeur sur un objet et un pointeur sur un tableau d'objet. Il faut lui indiquer dans la syntaxe de `delete` qu'il s'agit d'un tableau. Dans ce cas, le destructeur est appelé pour chaque élément du tableau avant que la mémoire ne soit libérée.

Il faut indiquer dans le programme l'instruction « `delete [] pt;` ». Dans cet exemple, le programme laisse, dans le meilleur des cas, de la mémoire perdue ou, dans le pire, cassera le chaînage du tas. L'erreur n'apparaîtra que très tardivement, lorsqu'il n'y aura plus de mémoire! Une version de debug de « `::new` » et de « `::delete` » peut la signaler en vérifiant la validité d'un pointeur avant d'effectuer un « `::delete` » (Voir « `::NEW` de debug », page 130). Si vous appelez « `delete pt` » à la place de « `delete [] pt` », le pointeur effacé n'est pas sur une adresse retournée par un « `::new` » précédent. Ceci est

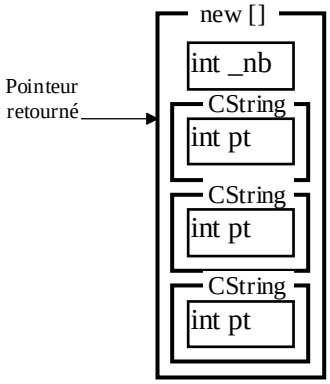
le résultat du compteur caché, ajouté par le compilateur. Cette particularité peut être détectée à l'exécution.

Dans les anciennes versions de C++, il fallait indiquer dans les crochets du `delete` le nombre d'éléments à effacer. Maintenant, depuis 1992, lors de la création du tableau, le compilateur mémorise le nombre d'éléments de celui-ci. Lors de l'effacement, le compilateur regarde ce nombre afin d'appeler correctement tous les destructeurs. Il n'est plus nécessaire d'indiquer le nombre d'éléments lors du `delete`, le compilateur n'en tient d'ailleurs plus compte.

La création et la destruction d'un tableau sont traduits à peu près comme ceci :

```
void main()
{ // CString* pt=::new CString[3];            Allocation du tableau
  CString* pt=
    (CString*)::new(sizeof(int)+           // Ajoute un entier au début
       sizeof(CString)*3);
  if (pt!=NULL)
  { *(int*)pt=3;                           // Nb d'éléments du tableau
    pt=(CString*)((int*)pt+1);             // Avance le pointeur
    for (int i=0;i<3;++i)                  // Appelle le constructeur
     pt[i].CString();                      // pour tous les éléments
  }
  // delete [] pt;                            Libération du tableau
  if (pt!=NULL)
  { int j=*((int*)pt-1);                   // Nb d'éléments du tableau
    for (int i=--j;i>=0;--i)               // Appelle le destructeur
      pt[i].~CString();                    // pour tous les éléments
    pt=(CString*)((int*)pt-1);             // Recule le pointeur
   ::delete(pt);                           // Efface la zone pointée
  }
}
```

Ce qui se présente en mémoire comme suit :



Il est de la responsabilité du programmeur d'appeler la bonne version de `delete` suivant l'objet manipulé, sinon, seul le premier destructeur est appelé.

Il est à noter que si une exception est générée à la construction d'un des éléments du tableau, le mécanisme de remontée de la pile appelle les destructeurs uniquement pour les objets déjà créés.

Si vous convertissez un pointeur lors de l'allocation, il faut faire de même lors de la destruction. Par exemple, il est courant d'allouer la taille nécessaire à plusieurs objets, et d'appeler le constructeur de celui-ci lors de l'utilisation de cette zone mémoire.

```
{ CObj *tab=(CObj*)new char[sizeof(CObj)*10]; // Reserve la place
  int i=0;

  // ...
  tab[i].CObj::CObj(); // Construction d'un des éléments du tableau

  delete [] tab;       // Erreur
}
```

Dans ce cas, l'exécution de `delete [] tab` est erronée, car le compilateur va appeler un nombre aléatoire de destructeurs des objets soi-disant présents dans le tampon. Ils n'ont pas tous été construits. Il faut corriger le programme comme cela.

```
for (int j=max-1;j>=0;--j)
{ tab[j].CObj::~CObj(); // Appel des destructeurs
}
delete [] (char*)tab;   // Effacement de la réserve
```

Toujours utiliser `delete []` pour les objets alloués avec un `new []`.

Règle CPP.DEB.3

```
void main()
{
  char* pt;

  pt=new char [10];
  // ...
  delete pt;
}
```

C'est incorrect, pourquoi ? (N.B.: `char` n'est pas un objet !)

Dans la nouvelle norme, il existe deux allocations : l'allocation d'un objet simple, et celle d'un tableau d'objets. Il faut appeler « `delete [] pt` » car rien n'indique qu'il y a

compatibilité entre « `operator ::delete(void*)` » et « `operator ::delete [] (void*)` ». Chaque allocateur, de tableau d'objet ou d'un seul objet, peut maintenir ses propres listes de mémoires utilisées ou vides. Le C++ offre trois allocateurs différents :

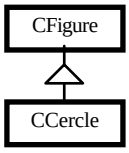
- `malloc` et `free`
- `new` et `delete`
- `new []` et `delete []`

Le compilateur optimise en général pour les objets de base en remplaçant dans ce cas le `delete []` par un `delete` simple, mais rien ne le garantit. La norme ANSI/ISO du C++ introduit également un opérateur `new []()` qui doit être associé au `delete []()`. Cet opérateur amplifie le risque d'erreurs entre les deux écritures.

Toujours effacer les tableaux de types standard par `delete []()`.

Règle CPP.DEB.4

```
class CFigure
{ public:
  virtual void trace()
  { cout << "CFigure::trace()" << endl;
  }
  ~CFigure()
  { cout << "CFigure::~CFigure()" << endl;
  }
};

class CCercle : public CFigure
{ public:
  virtual void trace()
  { cout << "CCercle::trace()" << endl;
  }
  ~CCercle()
  { cout << "CCercle::~CCercle()" << endl;
  }
};
```




```
void main()
{
  CFigure* p;
  p=new CFigure();
  p->trace();
```

```
  delete p;
  p=new CCercle();
  p->trace();
  delete p;
}
```

Qu'affiche `main`, pourquoi et comment corriger ?

`main` affiche :

```
CFigure::trace()
CFigure::~CFigure()
CCercle::trace()
CFigure::~CFigure()
```

Le destructeur de `CCercle` n'est pas appelé ! `p` est un pointeur de type `CFigure`. L'appel de `delete` appelle le destructeur de `CFigure`. Le destructeur est une méthode d'instance. Le compilateur ne peut pas savoir quel est la classe de l'instance pointée. Il faut qu'il utilise le polymorphisme pour résoudre cela. Le polymorphisme n'existe qu'avec les méthodes virtuelles. Pour avoir un fonctionnement correct, il faut déclarer le destructeur de `CFigure` en `virtual`. Dans ce cas, le destructeur de `CCercle` le devient aussi et tout rentre dans l'ordre. Par principe:

Toujours déclarer le destructeur en `virtual` pour les classes ayant une méthode virtuelle.

Cela n'ajoute aucun octet à l'objet. Seul le tableau des fonctions virtuelles augmente (Voir « Héritages », page 265).

Règle CPP.DEB.5

```
class CHopital
{ public:
  void afficheNom()
  { cout << "CHopital::afficheNom()" << endl;
  }
};
```


```
void main()
{
  void (*ptf)(void);                        // Pointeur de fonction

  ptf=&CHopital::afficheNom;
  ptf();                                    // Appel de CHopital::f()
}
```

Ce n'est pas compilable, pourquoi et comment corriger ?

`CHopital::afficheNom()` est une fonction membre, et non une fonction simple. Le compilateur ajoute le paramètre caché `this` pour cette fonction. Ce n'est pas compatible avec un simple pointeur de fonction. Il faut dans ce cas, soit déclarer la méthode `CHopital::afficheNom()` en `static`, soit utiliser les « pointeurs de membre » du langage.

La méthode `CHopital::afficheNom()` est générée comme cela :

```
void CHopital::afficheNom(CHopital * const this)
{ cout << "CHopital::afficheNom()" << endl;
}
```

Lors de l'appel d'une méthode, il faut fournir le pointeur `this`. Les pointeurs de membres sont là pour ça. Ils sont toujours associés à un type d'objets. Ils peuvent être utilisés pour tous les membres, attributs ou méthodes. En fait, ils possèdent une sorte d'offset sur l'élément de l'objet. Pour plus d'explications sur les pointeurs de membres, consultez le paragraphe 8.1c de [Stroustrup, ARM:94].

La solution à l'aide des pointeurs de membres est la suivante :

```
void main()
{ void (CHopital::*ptf)(void);
  ptf=&CHopital::afficheNom;

  CHopital hopital;                          // Il faut une instance
  (hopital.*ptf)();
}
```

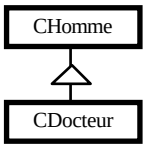
Il faut utiliser une instance pour appeler la méthode pointée par `ptf`.

Il existe trois algorithmes pour appeler une méthode. Si celle-ci est simple, un appel de fonction suffit. Si celle-ci est virtuelle, il faut passer par la table de sauts. Et enfin, si elle est virtuelle pour une classe héritée virtuellement, il faut retrouver la table de sauts avant l'appel (Voir « Héritages », page 265).

Un pointeur de membre pouvant accéder à tout type de méthode, le troisième algorithme doit être utilisé afin de fonctionner quelle que soit la méthode pointée. Certains compilateurs possèdent des options permettant de préciser quel type de méthode utiliser pour un pointeur de membre donné. Cela permet de générer un code nettement plus rapide et plus court.

Toujours utiliser les pointeurs de membres pour accéder dynamiquement à une méthode d'un objet.

Règle CPP.DEB.6




```
class CHomme
{ int age;
  public:
  virtual void trace() const
  { cout << "CHomme::trace()" << endl;
  }
  virtual ~CHomme()
  {}
};

class CDocteur : public CHomme
{ const char* diplome;
  public:
  virtual void trace() const
  { cout << "CDocteur::trace()" << endl;
  }
};

void f(const CHomme& x)
{ x.trace();
}

void g(const CHomme x)
{ x.trace();
}

void main()
{ CDocteur docteur;
  f(docteur);
  g(docteur);
}
```

Qu'affiche `main`, pourquoi ?

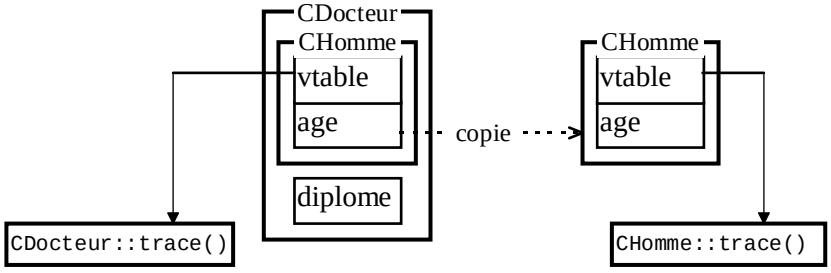
`main` affiche :

```
CDocteur::trace()
CHomme::trace()
```

On imagine souvent qu'une référence constante est strictement équivalente à l'objet lui-même. On utilise souvent cette écriture pour accélérer le programme, pour éviter de copier l'objet. Mais, dans certains cas, il y a une différence. En effet, dans l'appel de `f()`, la référence pointe sur l'objet `docteur`. Dans ce cas l'appel à la méthode virtuelle est celle de `CDocteur`.

Pour la fonction `g()`, l'appel est différent. Le paramètre est passé par valeur. Dans ce cas, il y a une copie de l'objet `docteur` dans un objet de type `CHomme`. L'appel de `g(docteur)` commence par appeler le constructeur de copie pour construire l'objet

paramètre de type `CHomme`. La copie d'un objet ne copie pas le pointeur sur la table virtuelle. C'est logique car la méthode `CDocteur::trace()` considère que le pointeur `this` pointe sur un objet de type `CDocteur` afin de pouvoir accéder à l'attribut `diplome`. Il n'existe pas d'attribut `diplome` dans un objet de type `CHomme`. Si le pointeur de la table virtuelle était copié, la fonction `CDocteur::trace()` accéderait à une zone mémoire invalide en utilisant l'attribut `diplome`. Le pointeur de la table de saut virtuel n'est jamais copié. Seuls les attributs de `CHomme` de l'objet de type `CDocteur` sont copiés.



L'objet `x` de la fonction `g()` contient une copie des éléments `CHomme` de `CDocteur`. L'appel à la méthode virtuelle est donc dans ce cas exécuté sur la méthode `CHomme::trace()`.

On utilise souvent une référence pour gagner de la place par rapport à une copie, mais dans certains cas, le fonctionnement peut être différent.

Il est à noter qu'une référence constante peut créer un objet temporaire là où une référence normale ne le fait pas.

```
long& r1=1;       // Erreur
const long& r2=2; // Création d'un objet temporaire de type long
```

Toujours utiliser des références pour les objets ayant des méthodes virtuelles.

Règle CPP.DEB.7

```
class CString
{ char* buf;
  public:
    CString(const char* str)
  { buf=(char*)::malloc(::strlen(str)+1);
    ::strcpy(buf,str);
  }
    CString(const CString& x)
  { buf=(char*)::malloc(::strlen(x.buf)+1);
    ::strcpy(buf,x.buf);
  }
    ~CString()
  { ::free(buf);
  }
  CString& operator =(const CString& x)
  { ::free(buf);
    buf=(char*)::malloc(::strlen(x.buf)+1);
    ::strcpy(buf,x.buf);
    return *this;
  }
};
```

Cette classe présente un risque d'erreur potentiel à l'utilisation. Dans quel cas et comment corriger ?

Le problème provient de l'opérateur d'égalité. Si celui-ci reçoit en paramètre lui-même, son tampon est effacé avant d'être utilisé dans la fonction `strlen()`.

```
void main()
{ CString a="abc";
  CString b="def";
  CString& ra=a;

  a=b;  // Pas de probleme
  a=a;  // Erreur
  a=ra; // Erreur
}
```

Cette erreur reste en général cachée ! Cela arrive essentiellement avec les références. Il faut ajouter au début de la méthode d'affectation la ligne :

```
if (this==&x) return *this;
```

afin de l'éviter. D'une manière générale, dans les opérateurs d'affectation, il faut traiter ce cas soit par la démarche précédente, soit par un `assert`, si cette situation ne doit pas se présenter.

Toujours ajouter, pour tout `operator =()`, au début de la méthode : « `if (this==&x) return *this;` » ou « `assert(this!=&x);` ».

Règle CPP.DEB.8

```
template <class T> class A
{ T m;
  public:
    A(const T z)
  { m=z; }
};
```

A<T>

Ce `template` peut échouer, pour quel type d'objet ?

Pour les classes, l'affectation et l'initialisation peuvent être différentes. Ce n'est pas conseillé, mais c'est possible. Il existe également une différence pour certains types de base. L'initialisation d'une référence est différente de l'assignation de celle-ci.

```
void main()
{ int  i=1;
  int& j=i; // Initialisation : 'j' se refere à 'i'
  j=2;      // Assignation : 'i' prend la valeur 2
}
```

Ce point peut être important pour la rédaction des `template` avec un type paramétrable. Il faut rédiger correctement le `template` pour qu'il fonctionne avec une référence.

Si vous utilisez le `template A` avec une référence, l'élément `m` n'est pas initialisé. Il faut modifier la déclaration comme ceci :

```
template <class T> class A
{ T m;
  public:
    A(const T z) : m(z)
  { }
};
```

Un autre cas est différent.

```
char str[]="abcd"; // Initialisation de str
void main()
{ str="cdef";      // Erreur: ecriture impossible
}
```

Pour être sûr qu'il y a concordance entre l'assignation et l'affectation, il existe une syntaxe permettant d'éviter de dupliquer le code du destructeur et du constructeur de copie dans l'assignation.


```
class CString
{ char* _str;
  public:
    CString()
  { _str=new char[10]; }
    CString(const CString& x)
  { _str=new char[10];
    memcpy(_str,x._str,10);
  }
    ~CString()
  { delete [] _str;
  }
  // Ecriture générique pour l'affectation
  CString& operator =(const CString& str)
  { if (&str!=this)
    { this->CString::~CString();            // Efface this
      new (this) CString(str);              // Constructeur de copie
    }
    return *this;
  }
};
```

Il faut un opérateur `::new` déclaré comme cela :

```
void* new(size_t s,void* x) { return x; }
```

Attention, l'appel du destructeur dans l'opérateur d'affectation n'appelle pas la version virtuelle, si elle existe, ce qui est voulu. Si une classe hérite de `CString`, l'appel de la version virtuelle du destructeur détruirait entièrement cette classe, alors que seule la partie `CString` de l'instance serait reconstruite. Les classes héritant de `CString` doivent déclarer l'opérateur d'affectation sans appeler la version de `CString`.

Toujours rédiger un `template` en pensant à un objet paramétrable du type référence.

Règle CPP.DEB.11


```
class CBorne
{ int maximum;
  int minimum;
  public:
    CBorne(int t) : minimum(t), maximum(minimum +1) {}
  void print() { cout << minimum << ',' << maximum << endl; }
};
void main()
```

```
{ CBorne borne=3;

  borne.print();
}
```

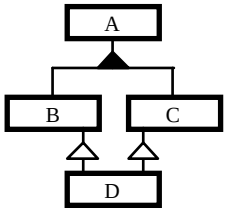
Qu'affiche `main`, pourquoi ?

`main` affiche `3` suivi de n'importe quoi. En effet, l'ordre des initialisations indiquées dans un constructeur n'a aucun effet. Les membres de l'objet `a` sont initialisés dans l'ordre de déclaration de la classe. L'appel de `maximum(minimum+1)` est effectué avant l'appel de `minimum(t)`. La valeur de `minimum`, à ce moment, est aléatoire. `maximum` prend donc une valeur quelconque. Ensuite, `minimum` est correctement initialisé. Les compilateurs ne signalent généralement pas ce problème. Des outils type « lint » permettent de le détecter.

Toujours initialiser les éléments dans l'ordre de déclaration dans l'objet.

Règle CPP.DEB.12

```
class A
{ public:
  int a;
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

void main()
{
  cout << "offset de a dans D " << offsetof(D,a) << endl;
}
```



La macro `offsetof` de la norme Ansi C permet d'obtenir l'offset d'un attribut dans une structure. Ce n'est pas exécutable, pourquoi ?

L'héritage virtuel est généré avec un pointeur caché dans les classes `B` et `C` (Voir « Héritages », page 265). Pour accéder à l'élément `a`, le compilateur passe par ce pointeur caché.

```
{ D d;
  d.a=3;
}
```

est traduit en :

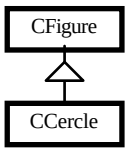
```
{ D d;
  d._ptA->a=3;
}
```

Pour connaître l'adresse de `a` dans l'objet `D`, le même mécanisme est mis en place. La conversion est résolue à l'exécution. Il faut posséder un objet valide pour obtenir l'adresse de `a`. La macro `offsetof` utilise l'adresse `NULL` comme base de pointeur sur le type `D`. Ensuite, elle soustrait l'offset de l'élément `a` de l'objet `D`. Pour cela, le compilateur recherche l'élément caché `_ptA` à l'adresse zéro. Cet élément n'existe pas. Le compilateur utilise alors une adresse aléatoire. C'est pour cela que cet exemple ne fonctionne pas. Il faut utiliser les pointeurs de membres pour obtenir cette fonctionnalité. Un pointeur de membre est assimilable à un offset d'un attribut. Mais en plus, des informations supplémentaires permettent de maîtriser les problèmes d'héritage virtuel et de méthodes virtuelles.

Toujours utiliser les pointeurs de membres plutôt que l'offset d'un attribut.

Règle CPP.DEB.14

```
class CFigure
{ public:
  virtual void trace()
  { cout << "CFigure::trace()" << endl;
  }
  // Constructeur
    CFigure()
  { trace();
  }
  virtual ~CFigure()
  {}
};

class CCercle : public CFigure
{ public:
  virtual void trace()
  { cout << "CCercle::trace()" << endl;
  }
};

void main()
{
  CFigure figure;
  CCercle cercle;
  cercle.trace();
}
```



Qu'affiche `main`, pourquoi ?

`main` affiche :

```
CFigure::trace()
CFigure::trace()
CCercle::trace()
```

Les fonctions virtuelles sont implantées à l'aide d'un pointeur caché dans l'objet référençant une table de sauts (Voir « Héritages », page 265). Ce pointeur est initialisé lors de l'accolade ouvrante du constructeur. Le constructeur de `CCercle` appelle le constructeur de `CFigure`, qui commence par initialiser le pointeur caché sur SA table de sauts. Il appelle ensuite `trace()`. Puis, l'accolade du constructeur de `CCercle` est exécutée. Le pointeur caché est écrasé par l'adresse de la table de sauts de `CCercle`. A partir de ce moment, l'appel de `trace()` est correct. C'est pour cela que l'appel direct de `trace()` pour l'instance `cercle` fonctionne. Il est généralement incorrect d'appeler une méthode virtuelle dans un constructeur. Certains compilateurs signalent ce risque par un `warning`.

Les constructeurs de `CFigure` et de `CCercle` sont générés comme cela :

```
void CFigure::CFigure(const CFigure * const this)
{ this->vptr=CFigure::vtable;
  this->vptr[0]();                          // Appel de trace
}

void CCercle::CCercle(const CCercle * const this)
{ CFigure::CFigure(this);
  this->vptr=CCercle::vtable;
}
```

Pour contourner le problème, il faut séparer la construction n'utilisant pas de méthode virtuelle, de la partie les utilisant. Une astuce consiste à déclarer un constructeur protégé identifié par un `enum` particulier pour pouvoir n'initialiser que la partie sans méthode virtuelle. Enfin, appeler la suite du constructeur. Cela peut se traduire comme suit :

```
class CFigure
{ protected:
  enum TGuru { Guru };                      // Id de constructeur
    CFigure(TGuru)
  { // Constructeur sans virtuel
    // ...
  }
  void ctrVirtual()
  { // Constructeur avec virtuel
    trace(); // Appel virtuel
  }
  public:
  virtual void trace()
  { cout << "CFigure::trace" << endl; }
    CFigure()                               // Appel sans virtuel
  { this->CFigure::CFigure(Guru);
```

```
    ctrVirtual();                          // Appel avec virtuel
  }
  virtual ~CFigure()
  {}
};

class CCercle : public CFigure
{ public:
  virtual void trace()
  { cout << "CCercle::trace" << endl; }
    CCercle() : CFigure(Guru)
  { CFigure::ctrVirtual();                 // Appel avec virtuel
  }
};
```

Toutes les classes dérivées doivent utiliser ce mécanisme, si elles désirent modifier les méthodes virtuelles utilisées par le constructeur. Pour l'extérieur de la classe, tout est normal. Il n'y a pas deux appels à effectuer pour construire l'objet. Celui-ci peut être construit dans une variable temporaire si nécessaire.

Les destructeurs procèdent de même. Le pointeur `vptr` est modifié avant l'exécution du destructeur.

Les destructeurs de `CFigure` et de `CCercle` sont générés comme cela :

```
void CFigure::~CFigure(const CFigure* const this)
{ this->vptr=CFigure::vtable;
  // code du destructeur
}

void CCercle::~CCercle(const CCercle* const this)
{ this->vptr=CCercle::vtable;
  // code du destructeur
  CFigure::~CFigure(this);                 // Appel du dtr de CFigure
}
```

Ne jamais appeler de méthodes virtuelles dans un constructeur ou un destructeur.

Règle CPP.DEB.15

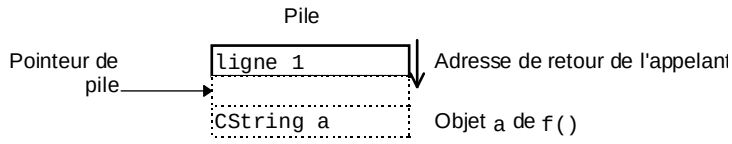
```
CString& f()
{ CString a="ab";
  return a;
}
CString& g()
{ CString a="/dir/";
  CString b="file.ext";
  return a+b;
}
```

```
void main()
{ cout << f() << endl;
  cout << g() << endl;                    // 1
}
```

Qu'affiche `main`, pourquoi et comment corriger ?

La fonction `f` cherche à renvoyer une référence sur une variable automatique. Celle-ci est détruite lors du retour de la fonction. Certains compilateurs signalent cette erreur, d'autres non. Par chance, cela peut fonctionner !

La pile lors du retour de la fonction `f()` se présente ainsi :



Le pointeur de pile devant pointer sur une zone mémoire vide, c'est équivalent d'écrire un code C du type :

```
int* f()
{ int a=3;
  return &a;
}
```

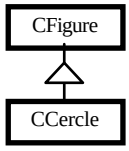
Si le programme appelle une fonction juste après l'appel de `f()` ou qu'une interruption matérielle est générée après le retour de la fonction, la valeur de `a` est modifiée !

La fonction `g()` retourne une référence sur un objet temporaire. Pour les raisons ci dessus, c'est impossible. Pour corriger le programme, il faut renvoyer un objet et non une référence.

Ne jamais renvoyer une référence sur une variable locale ou un paramètre.

Règle CPP.DEB.16

```
class CFigure
{ public:
  virtual ~CFigure()=0;
};

class CCercle : public CFigure
{ public:
```



```
  ~CCercle()
  {}
};
```

Ce n'est pas compilable, pourquoi et comment corriger ?

La réponse est assez simple. Un destructeur appelle toujours le destructeur de sa classe de base. Le destructeur `CCercle::~CCercle()` appelle `CFigure::~CFigure()`. Cette méthode n'existe pas car elle est déclarée en virtuelle pure. Les compilateurs devraient signaler cette erreur lors de la compilation, mais la plupart la signalent lors du `link` par l'absence de la méthode `CFigure::~CFigure()`.

Si vous désirez qu'il ne soit pas possible de créer d'instances d'un objet, déclarez les constructeurs en `protected`.

Ne jamais déclarer de destructeur en `virtual` pur.

Règle CPP.DEB.17

```
void main()
{
  char* pt;

  pt=new char [10];
  // ...
  pt=(char*)realloc(pt,sizeof(*pt)*20);
  // ...
}
```

C'est incorrect, pourquoi ?

Rien ne garantit qu'un pointeur alloué *via* `new` est compatible avec `malloc`, `free`, ou `realloc`. Il n'est donc pas possible d'appeler `realloc`. Dans le concept « Objet », il est difficile d'envisager un `realloc` car cela peut déplacer les objets en mémoire ce qui invaliderait toutes les références aux objets. Il faudrait ajouter un opérateur de « déplacement » d'un objet, différent d'un constructeur de copie. La sémantique n'est pas la même. Ce ne serait, par ailleurs, pas suffisant. Le modèle objet exige que chaque objet soit identifié de façon unique. En C++, l'identification d'un objet s'effectue avec l'adresse de celui-ci. Déplacer un objet en mémoire revient à changer d'objet (Voir « Classe mutante », page 93). C'est pour cela qu'il n'existe pas d'opérateur C++ équivalent au `realloc`. Malheureusement, dans la plupart des compilateurs, cette écriture fonctionne. Le `new` est associé au `malloc`. Mais, si vous utilisez une version de debug de `::new()`(Voir « `::NEW` de debug », page 130), celle-ci ajoute en général des informations autour des données allouées. Le pointeur retourné n'est alors plus compatible avec le `malloc`. Cela

peut entraîner un problème de portabilité. Votre programme peut fonctionner alors avec un compilateur, mais pas avec un autre.

Ne jamais mélanger les allocations *via* `::malloc` et celle *via* `::new`.

Règle CPP.DEB.18

```
template <class T>
class A
{ public:
  void f(int x);
  void f(T   x);
};
```

Cela ne fonctionne pas dans tous les cas. Pourquoi ?

Si l'utilisateur désire utiliser le `template` avec le type `int`, deux méthodes se retrouvent avec la même signature. Ceci est rejeté par le compilateur.

Ne jamais utiliser de surcharge dans un `template` avec le même nombre de paramètres pour les déclarations de méthodes.

Règle CPP.DEB.19

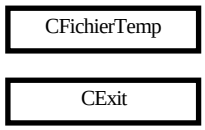
```
class CFichierTemp
{ FILE* st;
  char  nom[255];
  public:
    CFichierTemp()
  { ::strcpy(nom,"abc.tmp");
    st=::fopen(nom,"w");
  }


    ~CFichierTemp()
  { ::fclose(st);
    ::unlink(nom);
  }
};

void f()
{ int a;
  // ...
  exit(0);
}
```

```
void main()
{ CFichierTemp tmp;
  f();
}
```

Il y a une erreur, laquelle ?

La fonction `exit()` sort du programme. Les destructeurs des objets locaux ne sont pas appelés. Les destructeurs des objets globaux, eux le sont. Dans le cas présent, le fichier "`abc.tmp`" n'est jamais détruit. Le destructeur de `CFichierTemp` n'est jamais appelé. Il faut corriger ce programme en utilisant les exceptions.

```
class CFichierTemp
{ FILE* st;
  char  nom[255];
  public:
    CFichierTemp()
  { ::strcpy(nom,"abc.tmp");
    st=::fopen(nom,"w");
  }
    ~CFichierTemp()
  { ::fclose(st);
    ::unlink(nom);
  }
};

class CExit
{ int ret;
  public:
    CExit(int x) : ret(x) {}
    operator int() const { return ret; }
};

void f()
{ int a;
  // ...
  throw CExit(0);
}

int main()
{ CFichierTemp tmp;
  try
  { f();
  }
  catch (CExit x)
  { return x;
  }
}
```



Ne jamais utiliser la fonction `exit()` en C++.

Règle CPP.DEB.20

CFigure

```
class CFigure
{ public:
    CFigure()
  { cout << "CFigure::CFigure()" << endl;
  }
    ~CFigure()
  { cout << "CFigure::~CFigure()" << endl;
  }
};


void f(const char* format,...)
{ va_list ap;

  va_start(ap,format);
  if (!strcmp(format,"CFigure"))
  { CFigure x=va_arg(ap,CFigure);
  }
  va_end(ap);
}

void main()
{ CFigure tmp;
  f("CFigure",tmp);
}
```

Qu'affiche `main` ?

`main` affiche :

```
CFigure::CFigure()
CFigure::~CFigure()
CFigure::~CFigure()
```

Il y a deux destructeurs de `CFigure` appelés à la place d'un seul. La macro `va_arg` a été conçue pour le C, pas pour le C++. Le compilateur garantit la destruction de tous les objets qu'il crée. La macro `va_arg` construit un objet *via* des artifices de conversions multiples, sans appel de constructeur. L'objet `x` est construit sans constructeur, alors que son destructeur est appelé.

Ne jamais utiliser les macros `va_start`, `va_arg` et `va_end` en C++.

Règle CPP.DEB.21

CFigure

```
class CFigure
{ public:
    CFigure()
```

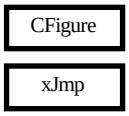
```
  { cout << "CFigure::CFigure()" << endl;
  }
    ~CFigure()
  { cout << "CFigure::~CFigure()" << endl;
  }
};

static jmp_buf env;

void f()
{ CFigure figure;
  longjmp(env,1);
}
void main()
{
  if (!setjmp(env))
  { f();
  }
  else
  { // ...
  }
}
```

Qu'affiche `main` ?

`main` affiche :

```
CFigure::CFigure()
```

Le destructeur n'est pas appelé. Il faut utiliser les exceptions à la place de `longjmp`.




```
class CFigure
{ public:
     CFigure()
  { cout << "CFigure::CFigure()" << endl;
  }
    ~CFigure()
  { cout << "CFigure::~CFigure()" << endl;
  }
};

class xJmp
{ int val;
  public:
  xJmp(int x) : val(x) {}
    operator int() { return val; }
};

void f()
{ CFigure figure;
  throw xJmp(1);
}

void main()
{
  try
```

```
  { f();
  }
  catch(xJmp x)
  { // ...
  }
}
```

Contrairement au `longjmp`, les exceptions garantissent de détruire tous les objets locaux présents dans la pile lors de la remontée de celle-ci.

Ne jamais utiliser `longjmp` en C++.

Règle CPP.DEB.22

```
#include <user.h>                          // utilise classe CUser

int comp(const CUser* val1,const CUser* val2)
{ return (*val1==*val2) ? 0 : (*val1<*val2 ? -1 : +1);
}

void main()
{ CUser tab[10];
  //...
  qsort(tab,10,sizeof(tab[0]),
        (int (*)(const void*,const void*))comp);
}
```

Ne connaissant pas `CUser`, ce code peut ne pas fonctionner, pourquoi ?

La fonction `qsort` déplace les objets en mémoire. Dans un modèle objet chaque objet doit avoir un identifiant unique. En C++, l'identifiant d'un objet est son adresse. Déplacer un objet en mémoire change l'identification de celui-ci. Toutes les références sur l'objet deviennent invalides.

La fonction `qsort` de la librairie ANSI C, déplace les objets par des copies binaires. Si des pointeurs référencent un des objets du tableau, ces pointeurs deviennent invalides. Si l'objet lui-même utilise une relation bidirectionnelle avec un autre objet, les pointeurs ne seront pas ajustés. N'étant pas sensé connaître l'implantation d'un objet, vous ne pouvez pas utiliser la fonction `qsort`. Il faut utiliser les constructeurs de copie pour obtenir un tableau d'objets triés, ou construire un tableau de pointeurs liés à chaque élément du tableau à trier.

```
int comp(const CUser* const * val1,const CUser* const * val2)
{ return (**val1==**val2) ? 0 : (**val1<**val2 ? -1 : +1);
}

void main()
{ CUser tab[10];
```

```
  //...
  CUser* tabsort[10];
  for (int i=0;i<10;++i) tabsort[i]=&tab[i];
  qsort(tabsort,10,sizeof(tabsort[0]),
        (int (*)(const void*,const void*))comp);
}
```

Ne jamais utiliser `qsort` avec des objets C++.

Règle CPP.DEB.25


```
class CRefEntier
{ public:
  int& ri;
    CRefEntier(int& i) : ri(i) {}
  CRefEntier& operator =(const CRefEntier& x)
  { if (this!=&x)
    { ri=x.ri; }
    return *this;
  }
};

void main()
{ int i=1;
  int j=2;
  CRefEntier i1(i);
  CRefEntier i2(j);
  i1=i2;
  cout << "i=" << i << ",j=" << j << endl;
}
```

Qu'affiche `main` ?

`main` affiche :

```
i=2,j=2
```

Si une classe possède une référence, il n'est pas possible d'écrire l'opérateur d'affectation. L'opérateur d'affectation par défaut n'est d'ailleurs plus présent. L'expression « `ri=x.ri` » modifie l'objet référencé, mais pas la référence elle-même. Pour résoudre ce problème, il faut utiliser l'écriture générique de l'opérateur d'affectation.

```
CRefEntier& operator =(const CRefEntier& x)
{ if (this!=&x)
  { this->CRefEntier::~CRefEntier();        // dtr
    ::new (this) CRefEntier(x);             // cctr
  }
  return *this;
}
```

Cette écriture appelle le constructeur, qui lui, peut initialiser la référence.

Il est à noter que les opérateurs de résolution sont obligatoires car il ne faut pas appeler les versions virtuelles des destructeurs. En effet, si une classe `CHandle` hérite de `CRefEntier`, et que `CRefEntier` possède un destructeur virtuel, l'appel de `this->~CRefEntier()` détruirait entièrement l'objet `CHandle`, alors que seule la partie `CRefEntier` de l'objet `CHandle` doit être détruite, pour être reconstruite à l'aide du constructeur de copie.

Toutes les classes dérivées doivent redéfinir leur opérateur d'affectation.

Éviter d'utiliser des références dans les classes !

Règle CPP.DEB.26

Fichier 1 :

```
class CCompteur1
{ public:
  int cnt1;
    CCompteur1()
  { cnt1=100;
  }
  void dec()
  { --cnt1;
  }
};

CCompteur1 GlobalCompteur1;
```

Fichier 2 :

```
extern CCompteur1 GlobalCompteur1;

class CCompteur2
{ public:
  int cnt2;
    CCompteur2()
  { GlobalCompteur1.dec();
    cnt2=GlobalCompteur1.cnt1;
  }
};

CCompteur2 GlobalCompteur2;

void main()
{ cout << "cnt1=" << GlobalCompteur1.cnt1 << endl;
```

```
  cout << "cnt2=" << GlobalCompteur2.cnt2 << endl;
}
```

Qu'affiche `main` et pourquoi ?

L'ordre d'initialisation des objets globaux entre plusieurs fichiers n'est pas défini. Suivant les compilateurs, l'initialisation de `GlobalCompteur1` peut être effectué avant celle de `GlobalCompteur2`, ou l'inverse. `main` affiche donc, suivant les cas :

```
cnt1=100
cnt2=-1
```

ou

```
cnt1=99
cnt2=99
```

Il n'existe pas de méthodes simple pour forcer l'ordre des initialisations. En général, les compilateurs offrent, *via* les `#pragma`, des moyens permettant de classer les initialisations. Il est par exemple impossible de garantir sur tous les compilateurs, la possibilité de faire une trace dans les flux standards lors des créations d'objets globaux. En effet, il est impossible de savoir si les objets globaux `cin`, `cout`, et `cerr` sont déjà créés lors de l'appel d'un constructeur d'objet global. Les compilateurs gèrent généralement une sorte de priorité pour l'initialisation de ces objets. C'est pour cela, qu'habituellement cela fonctionne. Les flux standard sont ainsi initialisés en premier. On peut également résoudre ce problème en mettant les objets globaux interdépendants dans le même fichier. Dans ce cas, l'ordre d'initialisation est celui de la déclaration des variables globales.

Les objets globaux ne sont pas liés lorsqu'ils sont présents dans une librairie. Seuls les objets globaux référencés par une méthode liée au programme sont ajoutés à la liste d'initialisation. Une simple déclaration à l'aide de `extern`, n'est pas suffisante pour inclure un objet global. Il faut l'utiliser en demandant par exemple son adresse. Un fichier « `.h` » peut forcer l'utilisation d'un objet global d'une librairie comme cela :

```
extern CObj theGlobalObj;
static CObj* pTheGlobalObj=&theGlobalObj;
```

Le pointeur `pTheGlobalObj` n'est jamais utilisé par l'application, il est créé dans chaque fichier incluant le fichier « `.h` ». Il est présent dans ce fichier afin d'indiquer au `link` qu'il est nécessaire d'initialiser l'objet `theGlobalObj` avant le `main`.

Jerry Schwarz [Stroustrup, ARM:94], est le premier programmeur des flux à utiliser une technique pour garantir l'initialisation des flux standard. Il a ajouté dans le « `.h` ».

```
class CIOStream_init
{ static unsigned count;
  public:
    CIOStream_init()
  { if (count++==0)
    { // ... Initialise les flux cin, cout et cerr
      init_flux();
    }
  }
    ~CIOStream_init()
  { if (--count==0)
    { // ... Destruction des flux cin, cout et cerr
    }
  }
};
static CIOStream_init IOStream_init;
```

Cela permet d'ajouter dans chaque fichier incluant un flux, un objet statique `IOStream_init` qui garantira l'initialisation des flux de base. Cet objet étant ajouté à tous les fichiers utilisant les flux, il garantit que ceux-ci sont initialisés en premier. Certains compilateurs permettent, par une option de compilation, d'éviter d'avoir un objet statique dans chaque fichier utilisant les flux. Pour cela, ils déclarent en commun, tous les objets statiques déclarés dans les fichiers « `.h` ». Dans ce cas de figure, une seule instance de `IOStream_init` est créée. La variable `count` n'est alors plus nécessaire.

Cette technique peut échouer si un constructeur appelle une fonction présente dans un autre fichier et que celle-ci utilise les flux. Si la variable statique est présente dans un fichier n'incluant pas le fichier « `iostream.h` », le constructeur peut être appelé avant l'initialisation des flux.

Fichier 1

```
// N'inclut pas fstream.h
void f(void);

class A
{ public:
  A() { f(); }                              // Appel de f()
} globalA;
```

Fichier 2

```
#include <iostream.h>
void f()
{ cout << "f" << endl; }
```

Ce cas de figure peut ne pas fonctionner. Il faut ajouter « `#include <iostream.h>` » dans le premier fichier.

Une autre approche consiste à déclarer une nouvelle version de l'opérateur `new`. L'opérateur `new` peut recevoir des paramètres supplémentaires comme tout type de méthode. Le problème est de signaler dans la syntaxe la valeur de ceux-ci. La norme autorise une écriture du type « `new (param) A;` » Les paramètres supplémentaires sont indiqués avant le constructeur de la classe. Dans le cas qui nous concerne, il faut déclarer un opérateur `new` renvoyant l'adresse indiquée dans le paramètre. Cela permet d'appeler le constructeur pour un objet à une adresse déjà connue.

```
void* operator new(size_t s,void* pt) { return pt; }
```

Cet outil est très souvent livré dans un fichier `.h` avec les compilateurs. Vous pouvez alors écrire une fonction initialisant dans l'ordre voulu les objets globaux. Cette fonction sera appelée au début du main.

```
A a=A::No_Init;
A b=A::No_Init;
void init_globaux()
{ new (&b) A("abc");
  new (&a) A("");
}
```

Bien sûr, il doit exister un constructeur ne faisant rien pour l'objet `A`. L'opérateur `new` particulier est un artifice pour pouvoir appeler le constructeur d'un objet. La nouvelle norme permettra un appel direct du constructeur. Dans ce cas, il suffira d'écrire :

```
void init_globaux()
{ b.A::A("abc");
  a.A::A("");
}
```

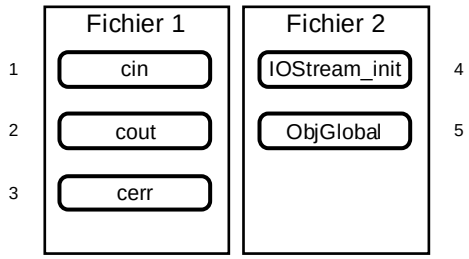
Cette écriture est plus explicite.

La fonction à appeler pour initialiser les flux standard peut par exemple être rédigée comme suit :

```
fstream cin=fstream::No_Init;
fstream cout=fstream::No_Init;
fstream cerr=fstream::No_Init;

void init_flux()
{ new (&cin)  fstream(1);
  new (&cout) fstream(2);
  new (&cerr) fstream(3);
}
void flush_flux()
{ cin.flush();
  cout.flush();
  cerr.flush();
}
```

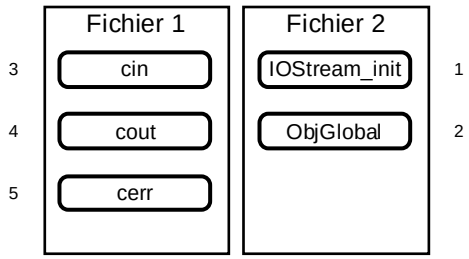
`init_flux` sera appelée lors du premier constructeur de `CIOStream_init`. `flush_flux` sera appelé lors du dernier constructeur de `CIOStream_init`. Pour que cela fonctionne, il faut que les destructeurs des trois flux de base ne fassent rien.

Deux situations peuvent alors se produire. Un objet global `ObjGlobal`, d'un autre fichier est construit après les constructeurs des flux standard. L'ordre d'initialisation est celui là :



Ceux-ci ont donc été construits mais pas initialisés. Le constructeur appelé a juste réservé la place mémoire et initialisé le pointeur de la table de saut. Lors de l'appel des constructeurs globaux du fichier contenant `ObjGlobal` le constructeur de `CIOStream_init` du module est appelé en premier. Celui-ci appelle la méthode `init_flux()`. Celle-ci va reconstruire correctement les flux standard. Ensuite, le constructeur de `ObjGlobal` va être appelé. Celui-ci peut utiliser les flux.

Une autre possibilité : le module contenant `ObjGlobal` est initialisé avant le module contenant les flux. L'ordre d'initialisation est celui-là :



Dans ce cas, le constructeur de `CIOStream_init` est appelé. Celui-ci appelle `init_flux()`. La fonction va initialiser pour la première fois les flux standard. Ils sont alors directement utilisables. Puis, le constructeur de `ObjGlobal` est appelé. Ensuite, les constructeurs des flux sont appelés. Ceux-ci ne faisant que modifier le pointeur sur la table de saut, ils ne modifient pas l'initialisation précédente de l'objet. L'objet `ObjGlobal`

utilise un flux n'ayant pas encore été officiellement initialisé du point de vue du compilateur.

Maintenant, les compilateurs ont évolués, et ce problème est géré par chaque implantation du C++ plus élégamment à l'aide de champs particuliers, gérés lors du `link`. Cette approche n'est malheureusement pas portable.

Le comité de normalisation a réfléchi à plusieurs techniques pour régler ce problème automatiquement. Une des approches étudiées consiste à ajouter dans les fichiers objets générés par le compilateur, un arbre de références croisées. Le linker peut alors parcourir cet arbre pour classer les objets globaux à initialiser. Finalement, ce problème n'a pas été résolu. La norme ANSI/ISO C++ indique qu'il n'y a pas d'ordre imposé pour l'initialisation d'objets `static` dans différents fichiers.

Éviter les interdépendances entre objets globaux.

Règle CPP.DEB.27

```
class CString
{ char buf[10];
  public:
    CString(const char* x)
  { strcpy(buf,x); }
  operator const char* () const
  { return buf; }
};

const char* Globalpt=CString("ABC");

void main()
{ cout << Globalpt << endl;
}
```

CString

Qu'affiche `main`, pourquoi ?

`main` affiche n'importe quoi ! Les objets globaux sont initialisés avant l'exécution de la fonction `main`. Les programmes C++ possèdent un `Startup` qui commence par initialiser tous les objets globaux, puis appelle le `main`, et enfin, détruit tous ces objets. On pourrait penser que les objets temporaires nécessaires à l'initialisation des objets globaux appartiennent à ce startup, il n'en est rien. Chaque initialisation est localisée dans une fonction distincte. Les objets temporaires sont présents uniquement dans ces fonctions. Lors de l'appel du `main`, ils ont déjà été détruits. Le compilateur crée un code du type :

```
const char* Globalpt;
void InitGlobalpt()
{ CString tmp("ABC");
```

```
  Globalpt=tmp.operator const char *();
  tmp.~CString();
}
void Startup()
{ // Initialise tous les objets globaux
  InitGlobalpt();
  // ...
  // puis appelle le main
  rc=main(argc,argv);
  // Détruit les objets globaux
  // ...
}
```

Le pointeur pointe sur l'objet `tmp` qui a déjà été détruit et n'est plus présent dans la pile lors de l'exécution du `main`. La version 1.5 de Microsoft bug sur ce point. Le destructeur de `CString` n'est jamais appelé bien que la pile soit libérée. C'est particulièrement trompeur car une écriture comme « `const char* Globalpt="ABC"` » fonctionne correctement. Dans ce cas, un tableau statique est créé possédant la chaîne de caractères, puis le pointeur `Globalpt` est ajusté pour pointer sur ce tableau. Toutes les chaînes de caractères constantes sont stockées dans des tableaux statiques.

Pour corriger le programme, il faut écrire :

```
const char* Globalpt=*new CString("ABC");
```

Dans ce cas, l'allocation de l'objet `CString` n'est pas effacée lors de la fin du programme.

On peut également supprimer l'utilisation de l'objet temporaire, en le rendant statique.

```
static CString tmp("ABC");
const char* Globalpt=tmp;
```

Il aurait peut-être été préférable de déclarer tous les objets temporaires nécessaires à l'initialisation des objets globaux en `static`, comme c'est le cas pour les constantes « chaînes de caractères ». Le comité de normalisation a tranché différemment.

Le seul cas où les objets temporaires, servant à initialiser les objets globaux, sont considérés comme `static` est l'initialisation des références.

```
const CString& RString=CString("abc");
```

déclare l'objet `CString("abc")` en statique. Le compilateur est certain que c'est le meilleur choix possible, car sinon, la référence deviendrait invalide lors du `main`.

Éviter d'utiliser des objets temporaires pour initialiser les objets globaux.

Règle CPP.DEB.31

```
void main()
{
  int i=5;

  cout << i + 3;
  cout << i & 3; // Erreur
}
```

Ce n'est pas compilable, pourquoi et comment corriger ?

C'est assez facile. Il ne faut pas oublier que l'utilisation des flux en C++ se fait par les opérateurs « << » et « >> ». Ceux-ci ont été choisis pour leur représentation symbolique signifiant « mettre dans le flux » ou « sortir du flux ». Ils représentent des flèches partant ou allant vers le flux. Initialement, il s'agit d'opérateurs de décalage binaire. Si vous regardez la table des priorités (Voir « Table de priorités », page 311), vous constaterez que la priorité du « plus » est supérieure à celle du « << ». Par contre, celle du « & » est inférieure. Le compilateur comprend « `(cout << i) & 3` » ce qui est incorrect. Cela peut également être traduit en :

```
(cout.operator <<(i)).operator &(3);
```

Il n'existe pas d'`operator &()` pour la classe `ostream`. La solution consiste à utiliser des parenthèses autour de « `(i & 3)` ». Cela permet d'avoir :

```
cout.operator <<(i & 3);
```

Lors de la création des flux, il aurait été possible de modifier l'opérateur « virgule » qui possède la plus basse priorité, à la place de l'opérateur « << ». Cet opérateur est moins symbolique, et il n'aurait pas été possible de différencier la lecture du flux, de l'écriture dans celui-ci. Un flux offrant la possibilité d'écrire et de lire avec le même objet n'aurait pas pu être construit. De plus, lors de l'écriture des premiers flux, en 1984, l'opérateur « virgule » ne pouvait pas être surchargé.

Utiliser les parenthèses pour encadrer les opérations lors des manipulations de flux.

Règle CPP.DEB.32

```
void f(char& r)
{ r=3; }

void main()
{ long l=0;
```

```
  f((char&)l);
  cout << l << endl;
}
```

Qu'affiche `main` ?

Le programme affiche une valeur différente suivant les compilateurs. En effet, si l'ordre des octets du microprocesseur commence par le poids faible, suivi du poids fort, le programme affiche « `3` ». Dans le cas contraire, l'affichage est incertain. Il n'est pas possible d'écrire partiellement dans une variable `long`. Ce programme peut être traduit en C comme ceci :

```
void f(char* r)
{ *r=3; }

void main()
{ long l=0;
  f((char*)&l);
  cout << l << endl;
}
```

Avec les microprocesseurs Motorola par exemple, ce programme fonctionne, mais pas avec la génération 80x86 d'Intel. Il est normalement incorrect d'exécuter une conversion avec une référence. Si un objet dérivé doit être utilisé *via* une référence sur l'objet de base, la conversion est automatiquement générée par le compilateur. La conversion explicite n'est alors pas nécessaire.

Toute conversion de référence est suspecte.

Règle CPP.DEB.33

En utilisant les classes presque standard `string`, `String`, `IString` ou `CString` :

```
void main()
{
  const char *pt;
  pt=CString("/dir/")+CString("file.ext");
  cout << pt;
}
```

Ces classes gèrent des tampons mémoire pour manipuler des chaînes de caractères. Avec ces outils, il n'est plus nécessaire de gérer les allocations mémoires des chaîne. Qu'affiche `main`, pourquoi ?

Sur la moitié des compilateurs, `main` affiche "`/dir/file.ext`". Sur l'autre moitié l'affichage est incertain. La nouvelle norme du C++ entraînera que l'affichage sera incertain. En effet, l'addition des deux `CString` est mise dans un objet temporaire. La norme

précédente indiquait que les objets temporaires étaient détruits *lorsque qu'ils n'étaient plus nécessaires*. Certaines implantations ont choisi de les détruire à la prochaine accolade fermante, d'autres, le plus tôt possible. La nouvelle norme indique que ces objets sont détruits lors du prochain « point virgule ». Donc, `pt` pointe sur le tampon de l'objet détruit, c'est-à-dire sur rien. Attention, cette erreur existe uniquement parce que l'opérateur « `operator const char *()` » est déclaré dans l'objet `CString`, ce qui renvoie un pointeur sur un des éléments de l'objet. Le programme est généré comme cela :

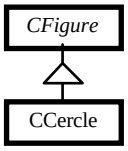
```
void main()
{
  const char *pt;
  CString tmp1;
  CString tmp2;
  CString tmp3;

  // Appel des constructeurs
  tmp1.CString("/dir/");
  tmp2.CString("file.ext");
  tmp3.CString(operator+(tmp1,tmp2));       // Ctr de copie

  const char* pt=tmp3.operator const char*();

  // Appel des destructeurs
  tmp3.~CString();                          // Effacement de la zone
                                            // pointé par pt !
  tmp2.~CString();
  tmp1.~CString();
  cout << pt;                               // Erreur !
}
```

Un objet temporaire est détruit à la fin de l'expression.

Règle CPP.DEB.34

Si on remplace dans le problème CPP.DEB.14 (Page 184) la structure `CFigure` par :

```
class CFigure
{ public:
  virtual void trace()=0;
    CFigure()
  { trace();
  }
  virtual ~CFigure();
};

// ...

void main()
{ CCercle cercle;
  cercle.trace();
}
```



Qu'affiche `main` et pourquoi ?

Le problème CPP.DEB.14 (Page 184) indique que le constructeur de `CFigure` appelle sa version de `trace`. Cette version n'existe pas ! C'est le seul cas où il est possible d'appeler une fonction virtuelle pure ! Les différentes implantations déclarent une fonction d'erreur indiquant, lors de l'exécution, qu'il y a eu appel d'une fonction virtuelle pure avant d'interrompre le programme. Si vous voyez apparaître ce message lors de l'exécution de votre programme, pensez immédiatement à cela !

Le compilateur peut détecter facilement l'appel d'une méthode virtuelle pure dans le constructeur et signaler l'erreur lors de la compilation. C'est plus difficile si le constructeur appelle une méthode, appelant elle-même une méthode virtuelle pure. Dans ce cas, il faut que le compilateur mémorise une table de références croisées pour détecter ce risque. C'est pour cela qu'en général, les compilateurs ne détectent pas ce problème, et préfèrent laisser l'exécution le signaler.

Si une méthode virtuelle pure est appelée à l'exécution, cela provient certainement de l'appel d'une méthode virtuelle dans un constructeur.

Règle CPP.DEB.35

L'objectif du problème suivant est de construire une classe simulant un pointeur qui verrouille et libère la mémoire utilisée *via* un Handle. Les OS comme Windows ou Mac OS possèdent une notion de Handle mémoire. La mémoire n'est plus référencée par un pointeur, mais *via* un Handle. Lorsque le programme désire utiliser la mémoire référencée, il faut auparavant bloquer cette mémoire pour obtenir un pointeur valide. Lorsque la zone mé-

moire n'est plus nécessaire, il faut débloquer cette mémoire. Cela permet d'implanter un « ramasse miettes ». C'est-à-dire la possibilité de réunir les zones vides du tas afin de mieux utiliser la mémoire. Deux appels successifs de `lock` pour le même handle ne renvoient pas le même pointeur mémoire mais toujours le même contenu. Cette approche peut également être utilisée pour augmenter artificiellement la mémoire disponible en utilisant un fichier temporaire. Lorsque le programme appelle un `lock`, le tampon correspondant est chargé depuis le fichier vers la mémoire. Lors d'un `unlock`, le tampon est recopié dans le fichier. L'inconvénient de ce type d'approche est que le programme doit scrupuleusement suivre les `lock`s et les `unlock`s afin de ne pas en oublier en chemin. La classe indiquée dans l'exemple suivant permet de s'affranchir de cela. La puissance du C++ permet de simuler un pointeur et de bloquer la mémoire utilisable lorsque cela est nécessaire.

```
// Gros objet à mettre dans un fichier temporaire ou ailleurs
class CBigObjet
{ public:
  char buf[10000];
    CBigObjet(char xFill) { ::memset(buf,xFill,sizeof(buf)); }
};

// Fonctions ::GetHandle, ::Lock et ::Unlock
typedef int HANDLE;
// ...

// Classe de simulation de pointeur sur CBigObjet
class CPtBigObjet
{ static HANDLE lockHandle;                 // Dernier handle bloqué
  HANDLE handle;
  CBigObjet* lock() const
  { ::Unlock(lockHandle);
    return (CBigObjet*) ::Lock(lockHandle=handle);
  }
  public:
    CPtBigObjet(HANDLE h)                : handle(h) { }
    CPtBigObjet(const CPtBigObjet& h)    : handle(h.handle) { }
  CBigObjet& operator *() const          { return *lock(); }
  CBigObjet& operator [](int i) const    { return lock()[i]; }
  CBigObjet* operator ->() const         { return lock(); }
  // ...
};

HANDLE CPtBigObjet::lockHandle=NULLHandle;

void main()
{ // Initialise via un handle
  CPtBigObjet pta=::GetHandle(sizeof(CBigObjet));
  CPtBigObjet ptb=::GetHandle(sizeof(CBigObjet));


  *pta=CBigObjet('A');
  ptb[0]=CBigObjet('B');
  pta->buf[1]='C';
```

```
  pta->buf[0]=ptb->buf[0];
}
```

Les syntaxes d'utilisations du pointeur dans `main` montrent que l'utilisation de l'objet est similaire à un pointeur. Pourtant, il y une erreur. Laquelle et comment la corriger ?

Le problème provient du fait qu'il est possible d'utiliser plusieurs objets bloqués simultanément. C'est le cas dans la ligne « `pta->buf[0]=ptb->buf[0]` ». Le pointeur retourné lors de l'appel à `ptb.operator ->()` n'est plus valide lorsque `pta.operator->()` est appelé. On ne peut pas savoir *a priori* combien de pointeurs peuvent être nécessaires simultanément, ni quand un pointeur n'est plus utilisé et peut donc être débloqué. Pour écrire correctement cette classe, il faut y ajouter une notion de LRU (Last Recent Use), afin de débloquer les pointeurs les plus vieux. Un LRU est un mécanisme maintenant la trace des utilisations d'une donnée. Lorsque certaines d'entre elles ne sont pas utilisées depuis un certain temps, le LRU décide de les supprimer de la mémoire. Le paramètre de ce LRU indiquera le nombre de pointeurs simultanément disponibles. Avec ce type d'implantation, tout fonctionne correctement. C'est toute la difficulté d'utiliser `operator ->()`. Celui-ci renvoie un pointeur « furtif ». Ce pointeur n'est valide qu'un certain temps. L'objet pointé peut disparaître de la mémoire et réapparaître ailleurs. Si l'on ne peut pas contrôler la validité de celui-ci, cet opérateur est rarement utilisable. Les quelques possibilités d'utilisation découlent de l'approche LRU décrite ci-dessus ; tester la valeur d'un pointeur pour détection d'erreurs, la rédaction des *pointeurs d'agrégations* (Voir « Durée de vie des objets », page 57) ou l'écriture des *smarts pointers* (Voir « Smart pointer », page 77). Une approche similaire a été choisie pour certaines implantations, au sein de l'ODMG (Object Database Management Group).

Ce problème existe aussi si vous désirez concevoir un conteneur type tableau avec cache sur disque où seuls les derniers éléments accédés du tableau sont en mémoire. Les autres sont copiés sur disque. Dans ce cas, l'`operator []()` retourne une référence « furtive ». Celle-ci n'est valide que lors de la présence dans le cache de l'élément du tableau .

Cet objet montre également un principe d'écriture ouvrant la porte à une évolution future d'un programme. Si vous déclarez un `typedef` pour tout pointeur d'objet, celui-ci peut, par la suite, évoluer vers l'approche ci-dessus sans changer les autres lignes du programme. Dans le cas contraire, il faut retrouver tous les « `X *` » pour les renommer avec le nom de la nouvelle classe. Par exemple, pour une classe `A`, déclarez un « `typedef A* PA;` » et n'utilisez que celui-ci pour pointer sur `A`. Par la suite, si vous constatez que vous n'avez pas assez de mémoire pour exécuter votre programme, remplacez le `typedef` en « `typedef CPtBigObjet PA;` ». Après recompilation de tout votre programme, celui-ci utilisera le disque dur pour accéder à ces données.

Pour rédiger un cache en surchargeant l'`operator ->()` utilisez un LRU (Last Recent Use).

Règle CPP.DEB.39

```
void f(long)   { cout << "f(long)" << endl; }
void f(char*)  { cout << "f(char*)" << endl; }

void main()
{ f(3);
  f(NULL);
}
```

Qu'affiche `main`, pourquoi ?

`main` affiche :

```
f(long);
f(long);
```

La valeur `NULL` est du type `int` ou `long`, suivant les compilateurs. Ce n'est plus comme en C ANSI, équivalent à `(void*)0`, car il n'est plus possible de convertir un pointeur `void` vers un pointeur d'un autre type (Voir « Différences entre le C et le C++ », page 307). Pour pouvoir comparer la valeur d'un pointeur avec la constante `NULL` il a fallu accepter une entorse au mécanisme de conversion. Seule la constante zéro peut être convertie implicitement en un pointeur.

Cela entraîne le refus d'utiliser la comparaison directe d'un pointeur. Les écritures du type :

```
if (p) // ...
```

ou

```
if (!p) // ...
```

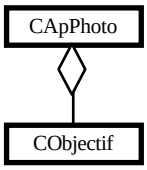
pour tester un pointeur avec la valeur `NULL`, ne sont pas valides. La conversion de la constante zéro vers un pointeur donne le pointeur `NULL`. Ce n'est pas forcément équivalent binairement à zéro. L'adresse `NULL` n'indique pas que la valeur du pointeur est à zéro. Toute valeur de pointeur peut être comparée avec la valeur `NULL`. La seule garantie décrite dans la norme ANSI C et C++ est qu'il n'existe pas d'objet présent à cette adresse. Une allocation mémoire ne peut pas retourner une adresse valide à l'adresse `NULL`. La comparaison d'un pointeur à l'aide de « `if (p)` » compare la valeur binaire de `p` mais pas si le pointeur `p` correspond à la valeur `NULL`. Dans la plupart des compilateurs, cela fonctionne, mais ce n'est pas portable. L'implantation courante de `NULL` en C ANSI est souvent `(void*)0`, mais cela n'est pas obligatoire. Un compilateur C peut très bien définir la valeur `NULL` comme égale à `(void*)0xFFFFFFFF`. Dans ce cas, le compilateur C++

convertit la constante zéro en `(void*)0xFFFFFFFF`. Certaines machines utilisent l'adresse zéro comme port électronique. L'écriture ou la lecture à cette adresse peut avoir des effets néfastes sur le matériel. Il n'est pas raisonnable sur ce type de machine d'avoir l'adresse `NULL` sur cette adresse. La valeur `NULL` n'est alors pas à zéro.

Ne pas surcharger une fonction ou une méthode avec un type numérique et un pointeur.

Règle CPP.DEB.40

```
class CApPhoto;
class CObjectif
{ public:
    operator CApPhoto() const;
};

class CApPhoto
{ CObjectif _objectif;
  public:
    CApPhoto(const class CObjectif& objectif)
  : _objectif(objectif) {}
};

inline CObjectif::operator CApPhoto() const
{ return CApPhoto(*this); }

void f(const CApPhoto& apphoto)
{ cout << "f" << endl;
}

void main()
{ CObjectif objectif;

  f(objectif); // Erreur
}
```



Ce n'est pas compilable, pourquoi ?

Il existe une ambiguïté lors de l'appel de `f(objectif)`. En effet, il existe deux chemins pour convertir un objet `CObjectif` en une référence constante sur un objet `CApPhoto`. Soit le compilateur construit une nouvelle instance de `CApPhoto` à l'aide de son constructeur recevant un objet `CObjectif`, soit le compilateur appelle l'opérateur de conversion de l'objet `CObjectif`. Ce type d'écriture peut fonctionner sans ambiguïté pendant un certain temps, mais une écriture particulière la fera apparaître par la suite.

Éviter les écritures ambiguës lors de la rédaction des conversions et des constructeurs.

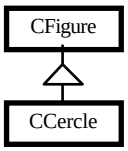
Règle CPP.DEB.41

```
class CObjNom
{ public:
  void* operator new(size_t,const char*);
};

void main()
{ CObjNom* p;
  p=new ("main") CObjNom();
  p=new CObjNom(); // Erreur
}
```

Ce n'est pas compilable, pourquoi ?

La norme indique que la surcharge d'une méthode perd la visibilité sur les versions héritées. Il faut redéfinir toutes les versions de la méthode surchargée.

```
class CFigure
{ public:
  void f(int i);
};

class CCercle : public CFigure
{ public:
  void f(unsigned u);
  void f(int i)
  { CFigure::f(i); } // Redefini
};
```



Cela évite d'appeler par erreur une méthode d'une classe de base lors d'un appel avec un mauvais paramètre. Si l'héritage est très profond, vous risquez, sans le savoir, de surcharger une méthode dont vous ignorez l'existence, et d'appeler une version inconnue lors d'un appel erroné, sans que le compilateur signale votre erreur.

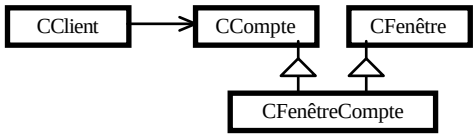
Pour les opérateurs, le mécanisme est le même. Si vous déclarez une version spécifique de l'opérateur new, il faut ajouter la version classique.

```
class CObjNom
{ public:
  void* operator new(size_t,const char*);
  void* operator new(size_t s)
  { ::operator new(s); }
};
```

Rien ne garantit que l'allocation d'un tableau de caractères est compatible avec l'allocation d'un seul objet (Voir règle CPP.DEB.3, page 174).

Ajouter toujours la syntaxe classique de l'opérateur `new` lors de la surcharge de celui-ci.

Règle CPP.DEB.42




```
class CCompte;
class CFenetreCompte;

class CClient
{ CCompte* _compte;
  public:
    CClient(CCompte* compte) : _compte(compte)
  { }
  void affiche();
};

inline CClient* createClient(CFenetreCompte* fenCpt)
{ return new CClient((CCompte*)fenCpt); } //1

class CFenetre
{ int _taille;
  //...
};

class CCompte
{ int _solde;
  public:
  virtual int solde() const
  { cout << "CCompte::solde" << endl;
    return _solde;
  }
  virtual ~CCompte()
  {}
};

class CFenetreCompte : public CFenetre,public CCompte
{
};
void CClient::affiche()
{ _compte->solde();
}

void main()
{ CFenetreCompte c;
  CClient* client=createClient(&c);
  client->affiche();
}
```

Ce n'est pas exécutable, pourquoi ?

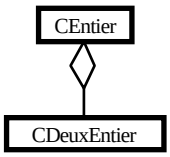
Les classes `CCompte` et `CFenetreCompte` sont déclarées « en avant ». Les déclarations complètes de ces classes se trouvent après les déclarations de la classe `CClient`. La ligne indiquée `1` fait une conversion explicite car le compilateur ne connaît pas les relations

entre les classes `CCompte` et `CFenetreCompte`. Il ne peut donc pas faire de conversion implicite. Le chapitre « Héritages » (page 265) du document indique qu'une conversion d'un pointeur vers une classe héritée en deuxième position modifie la valeur du pointeur. Dans l'exemple précédent, le compilateur ne connaît pas encore la relation d'héritage entre `CCompte` et `CFenetreCompte`. C'est pour cela qu'une conversion explicite est nécessaire. Si cette relation était connue, le compilateur pourrait générer une conversion implicite. Dans le cas présent, comme la relation n'est pas connue, le compilateur ne modifie pas la valeur du pointeur. L'appel de `solde()` dans la méthode `CClient::affiche()` utilise le pointeur caché `vptr` de `CCompte`. Celui-ci est incorrect car le pointeur converti ne pointe pas sur la partie `CCompte` de `CFenetreCompte`, mais sur la partie `CFenetre` de `CFenetreCompte`. L'appel de `affiche()` ne peut pas être exécuté. Si vous êtes obligé de rédiger une conversion explicite alors qu'une conversion implicite pourrait fonctionner, il faut modifier votre programme pour supprimer les déclarations « en avant ».

Ne pas déclarer des classes « en avant » si elles héritent l'une de l'autre.

Règle CPP.OPT.1

```
class CEntier
{ int _n;
  public:
    CEntier()
  { cout << "CEntier::CEntier()" << endl;
  }
    CEntier(int n)
  { _n=n;
    cout << "CEntier::CEntier(int)" << endl;
  }
  void operator=(int n)
  { _n=n;
    cout << "CEntier::operator=(int)" << endl;
  }
};

class CDeuxEntier
{ CEntier   _a;
  int       _b;

  public:
    CDeuxEntier()
  { _a=4;
    _b=5;
  }
};

void main()
```



```
{ CDeuxEntier c;
}
```

Qu'affiche `main`, pourquoi ?

`main` affiche :

```
CEntier::CEntier()
CEntier::operator=(int)
```

En effet, le créateur vide de `CEntier` est appelé. Il est préférable de toujours appeler le créateur de la classe de base avant l'accolade ouvrante d'un créateur. Cela évite de construire un objet pour immédiatement détruire ces valeurs par un opérateur d'affectation. Ceci est vrai également pour les éléments n'étant pas des objets comme `_b`. En effet, si plus tard vous modifiez le type de `_b`, le problème d'optimisation indiqué au-dessus se représente. Il faut écrire le constructeur de `CDeuxEntier` comme cela :

```
  CDeuxEntier() : _a(4), _b(3) {}
```

Toujours initialiser les éléments d'un objet en dehors des accolades du constructeur.

Règle CPP.OPT.2

```
class CEntier
{ int i;
  public:
    CEntier(int z=0)
  { i=z; }

 int operator ++()
  { i++;
    return i;
  }
};

void main()
{ CEntier a;

  a++; // Erreur
}
```

CEntier

Ce n'est pas compilable, pourquoi ?

La nouvelle norme indique qu'il y a maintenant une différence entre les opérateurs postfixes et suffixes. Il faut indiquer `++a` et non `a++` pour appeler la version déclarée dans l'objet. Pour accepter cette syntaxe, il faut ajouter la méthode « `operator ++(int)` ». L'opérateur suffixe oblige à manipuler une copie de l'objet afin de renvoyer la valeur de

celui-ci avant l'incrémentation. Cela prend du temps et n'a d'intérêt que si l'appelant utilise cette valeur.

```
CEntier CEntier::operator ++(int)
{ CEntier last=*this;
  ++*this;
  return last;
}
```

Si on désire uniquement incrémenter celui-ci, il est préférable d'écrire `++i`. Il est recommandé, de ne plus utiliser l'approche `i++` si ce n'est pas nécessaire, mais plutôt `++i`, même dans les boucles, car le type de l'objet d'index peut, par la suite, évoluer.

Toute utilisation d'un opérateur suffixe doit être justifiée par l'utilisation de l'objet avant le traitement de l'opérateur.

Règle CPP.OPT.4

```
class CObj
{ public:
    CObj()
    { cout << "CObj::CObj()" << endl; }
    CObj(const CObj&)
    { cout << "CObj::CObj(const CObj&)" << endl; }
};

CObj f() { return CObj(); }

CObj g()
{ CObj rc=f();
  return rc;
}
```

CObj

L'appel d'un constructeur de copie peut être évité, comment ?

La fonction `g()` doit être modifiée ainsi :

```
CObj g()
{ return f();
}
```

Il faut éviter de copier un objet dans différents intermédiaires. Vous ne savez pas forcément le temps que prend la copie d'un objet, ni la place mémoire nécessaire. Un objet peut posséder un tableau de 10.000 éléments. Les copies successives consomment inutilement la mémoire, et ralentissent le programme.

Renvoyer directement le retour d'une fonction pour éviter la création d'objets temporaires.

Règle CPP.OPT.5

CEntier

```
class CEntier
{ int a;
  public:
    CEntier()
  { cout << "CEntier::CEntier()" << endl; }
  CEntier& operator =(const CEntier&)
  { cout << "CEntier::operator =()" << endl;
    return *this;
  }
  CEntier& operator =(int x)
  { a=x;
    return *this;
  }
};

void main()
{ CEntier a1;
  a1=1;
  // ...
  CEntier a2;
  a2=a1;
}
```

Qu'affiche `main`, pourquoi ?

`main` affiche :

```
CEntier::CEntier()
CEntier::CEntier()
CEntier::operator =()
```

Le constructeur est toujours appelé. Si vous désirez modifier l'objet immédiatement après sa construction à l'aide d'une affectation, utilisez le constructeur de copie.

Toujours préférer l'initialisation à la place de l'assignation.

Règle CPP.OPT.6

xMsg

```
class xMsg
{ public:
    xMsg() {}
    xMsg(const xMsg&)
  { cout << "xMsg::xMsg(const xMsg&)" << endl; }
  virtual void print() const
  { cout << "print" << endl; }
  virtual ~xMsg()
  {}
```

```
};

void f()
{ throw xMsg();
}

void main()
{
  try
  { f();
  }
  catch(xMsg x)
  { x.print();
  }
}
```

Qu'affiche `main`, pourquoi ?

`main` affiche :

```
xMsg::xMsg(const xMsg&)
xMsg::xMsg(const xMsg&)
print
```

Une exception copie toujours l'objet ! Si vous capturez celle-ci par valeur, une deuxième copie est effectuée.

Il faut toujours recevoir une exception par référence pour accéder à la copie faite par le compilateur.

Envoyer (`throw`) les exceptions par valeur et les capturer (`catch`) par référence.

Règle CPP.OPT.7

```
void f(int value1,int value2=0)
{ if (value2)
  { // ...
  }
  else
  { // ...
  }
}
```

Comment rédiger la fonction `f()` pour l'optimiser ?

Si une méthode ou une fonction modifie profondément son comportement suivant la valeur d'un de ces paramètres, il est plus judicieux de découper la méthode en deux versions. L'écriture précédente n'est correcte que si le paramètre `value2` peut être appelé avec une

variable ayant la valeur zéro à l'exécution. Sinon, si l'appel ne s'effectue qu'à l'aide de constante, les deux écritures de la fonction sont préférables.

```
void f1(int value1)
{ // ...
}
void f2(int value1,int value2)
{ assert(value2!=0);
  // ...
}
```

Une fonction ne doit pas exécuter un code fondé sur la valeur d'un argument (défaut ou non) s'il n'est pas valorisé à l'exécution.

Règle CPP.OPT.8

```
class CComplex
{ protected:
  float reel;
  float imma;
  public:
    CComplex();
    CComplex(float xreel,float ximma);
    CComplex(const CComplex& x);
  // ...
  CComplex& operator =(const CComplex& x);
  CComplex operator +(const CComplex& x);
  CComplex& operator +=(const CComplex& x);
};


CComplex CComplex::operator +(const CComplex& x)
{ return CComplex(reel+x.reel,imma+x.imma);
}

CComplex& CComplex::operator +=(const CComplex& x)
{ *this=*this+x;
  return *this;
}
```

CComplex

Les opérateurs « + » et « += » ne sont pas optimisés, comment les améliorer ?

Que fait le compilateur pour compiler la méthode `CComplex::operator +()` ? Regardons le code généré :

```
void CComplex::operator +(CComplex* _ret,const CComplex& x)
{ CComplex tmp;
  tmp.CComplex::CComplex(reel+x.reel,imma+x.imma);// add,ctr
  _ret->CComplex::CComplex(tmp);                  // cctr
```

```
  tmp.CComplex::~CComplex();                    // dtr
}
```

Il y a un constructeur avec addition, un constructeur de copie, et un destructeur. La méthode `CComplex::operator +=()` est générée comme cela :

```
CComplex& CComplex::operator +=(const CComplex& x)
{ CComplex tmp;
  CComplex::operator +(&tmp,x);             // add, ctr, cctr, dtr
  this->CComplex::operator =(tmp);          // Affectation
  tmp.CComplex::~CComplex();                // dtr
  return *this;
}
```

On constate qu'il y a beaucoup de traitements intermédiaires pour obtenir le résultat.

Certains compilateurs détectent l'utilisation d'un objet temporaire utilisé pour le retour de la fonction, ils l'éliminent alors, pour utiliser directement l'objet retourné. Dans ce cas, la compilation de l'`operator +()` s'effectue comme suit :

```
void CComplex::operator +(CComplex* _ret,const CComplex& x)
{ _ret->CComplex::CComplex(reel+x.reel,imma+x.imma);   // add, ctr
}
```

La compilation de `operator +=()` n'exécute alors qu'une addition, un constructeur, une affectation, et un destructeur.

Il ne faut pas avoir d'effet de bord dans un constructeur de copie, car le compilateur peut optimiser le code et supprimer un appel.

Une autre approche consiste à utiliser l'`operator +=()` dans l'`operator +()`.

```
CComplex& CComplex::operator +=(const CComplex& x)
{ reel+=x.reel;
  imma+=x.imma;
  return *this;
}
CComplex CComplex::operator +(const CComplex& x)
{ return CComplex(*this)+=x;
}
```

Que fait le compilateur pour compiler la méthode `operator +()`?

Regardons le code généré :

```
void CComplex::operator +(CComplex* _ret,const CComplex& x)
{ CComplex tmp(*this);                      // cctr
  tmp.operator +=(x);                       // Additions
  _ret->CComplex::CComplex(tmp);            // cctr
  tmp.CComplex::~CComplex();                // dtr
}
```

L'`operator +=()` ne fait que deux additions. L'`operator +()` appelle un constructeur de copie pour créer un objet temporaire, effectue deux additions, rappelle une nouvelle fois le constructeur de copie, puis appelle le destructeur de l'objet temporaire. Cela donne suivant les versions :

v1	Addition	Constructeur	Affectation	Destructeur
v2	Constructeur de copie	Addition	Constructeur de copie	Destructeur

Le constructeur de copie est équivalent au constructeur normal. L'affectation est équivalente à un destructeur, suivi d'un constructeur. Dans ce cas précis, le destructeur ne fait rien. Donc, l'affectation est équivalente à un constructeur simple. Les deux versions sont équivalentes, mais la deuxième est plus rapide si le destructeur de l'objet a un coût non nul. De plus, ces versions sont équivalentes si et seulement si, le compilateur optimise l'`operator +()`. La deuxième version est généralement plus rapide.

On constate que pour obtenir le meilleur résultat, il faut rédiger distinctement les deux opérateurs.

```
CComplex CComplex::operator +(const CComplex& x)
{ return CComplex(reel+x.reel,imma+x.imma);
}
CComplex& CComplex::operator +=(const CComplex& x)
{ reel+=x.reel;
  imma+=x.imma;
  return *this;
}
```

Cette approche est la plus rapide. Il n'y a pas d'objet temporaire manipulé. Malheureusement, il est fort possible qu'il y ait dissonance sémantique entre les deux versions. Une erreur corrigée dans un des opérateurs ne le sera pas forcément dans l'autre. Si la performance est vraiment critique, il faut rédiger individuellement les deux versions. Si la maintenance est prioritaire, il faut

Utiliser `operator +=()` pour écrire `operator +()`.

Règle CPP.POR.1

```
class CVide
{
};

void main()
{ cout << sizeof(CVide) << endl;
  cout << sizeof('A') << endl;
}
```

Qu'affiche `main` ?

`main` affiche un nombre différent de zéro pour `sizeof(CVide)`. En effet, cela permet de garantir que les objets ont tous une adresse différente.

```
class CVide {};
static CVide a;
static CVide b;
static CVide* pA=&a;

void main()
{ if (pA!=&b) ...
}
```

De plus, un tableau d'objets `CVide` peut être parcouru par un pointeur. Cela peut, suivant les compilateurs avoir un effet bizarre. Pour les classes suivantes :

```
class CVide1 {};
class CVide2 {};
class CVide3 : public CVide1, public CVide2 {};
```

on *peut* avoir :

```
sizeof(CVide3)==sizeof(CVide1);
sizeof(CVide3)==sizeof(CVide2);
sizeof(CVide3)!=sizeof(CVide1)+sizeof(CVide2);
```

Certains compilateurs n'ajoutent pas d'octets lors de l'addition de plusieurs classes vides. Ils calculent la taille que devrait avoir réellement l'objet, et, si celle-ci est égale à « zéro », il l'ajuste à « un ». Il est malgré tout possible de demander une allocation de zéro octet. Dans l'exemple suivant :

```
void main()
{ char* p1=new char[0];
  char* p2=new char[0];
  assert(p1!=p2);
}
```

la norme garantit que l'allocation de plusieurs objets de taille nulle sont à des adresses différentes.

D'autre part, en C++, `sizeof('A')` est égale à `1` et non à `sizeof(int)` comme en C ANSI. Cette modification par rapport au C a été introduite lors de la création des flux. En effet, auparavant, une écriture comme « `cout << 'X';` » affichait « `88` » et non le caractère. Il a fallu différencier les entiers des constantes caractères pour régler le problème. Cela rend le langage plus cohérent. Il est à noter qu'il existe maintenant trois types de `char`. Le `char` sans attribut, le `signed char` et le `unsigned char`. Il s'agit de

trois types distincts permettant de différencier l'appel d'une fonction. C'est le seul type de base pouvant avoir trois versions !

Ne pas considérer le type `char` comme un type `int`.

Règle CPP.POR.2

```
void main()
{
  for (int i=0;i<10;++i)
    ;
  if (i==10) cout << "Fin de boucle" << endl;
}
```

Ce n'est pas compilable, pourquoi ?

Dans la future norme du C++, les objets déclarés dans les expressions ne sont visibles que dans celles-ci.

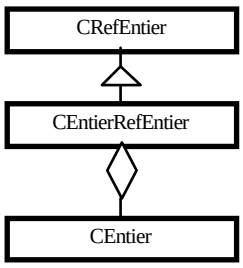
Il faut dorénavant écrire :

```
void main()
{
  int i=0;
  for (;i<10;++i)
    ;
  if (i==10) cout << "Fin de boucle" << endl;
}
```

Si une variable doit être utilisée après une boucle, la déclarer en dehors de celle-ci.

Règle CPP.POR.3

```
class CEntier
{ public:
  int i;
    CEntier(int z)
  : i(z) {}
};

class CRefEntier
{ CEntier& ra;
  public:
    CRefEntier(CEntier& x)
  : ra(x)
  { cout << "i="  << ra.i << endl;
```



```
  }
};

class CEntierRefEntier : public CRefEntier
{ CEntier a;
  public:
    CEntierRefEntier()
  : a(3), CRefEntier(a)
  {}
};

void main()
{ CEntierRefEntier c;
}
```

Qu'affiche `main`, pourquoi ?

`main` affiche n'importe quoi. En effet, le constructeur de la classe de base est appelé avant le constructeur des membres d'une classe. Dans ce cas, `CRefEntier(a)` est appelé avant l'initialisation de `a`. Dans le constructeur de `CRefEntier`, l'utilisation du paramètre `x`, cherche à afficher un élément de `CEntierRefEntier::a` qui n'est pas encore initialisé.

La solution consiste à modifier la classe `CEntierRefEntier` comme cela :

```
class CEntierRefEntier : public CRefEntier
{ CEntier* a;
  public:
    CEntierRefEntier()
  : CRefEntier(*(a=new CEntier(3)))
  {}
    ~CEntierRefEntier()
  { delete a; }
};
```

Utiliser un pointeur à la place d'un objet `CEntier` ; Initialiser ce pointeur dans l'appel du constructeur de `CRefEntier` ; Envoyer à `CRefEntier` la valeur du pointeur ainsi créée ; Effacer ce pointeur lors de la destruction de l'objet `CEntierRefEntier`.

Évitez d'utiliser un paramètre reçu par référence dans un constructeur.

autrement, le signaler dans les commentaires.

Règle CPP.STY.6

```
class CEntier
{ int a;
```

```
  public:
    CEntier(int x) { a=x; }
  int valeur() { return a; }
};

void fn(const CEntier& a)
{ cout << a.valeur() << endl;              // Erreur
}

void main()
{ CEntier a=3;
  fn(a);
}
```

Ce programme ne se compile pas, pourquoi ?

Le compilateur ne peut pas compiler car `fn` reçoit une référence constante et la méthode `valeur()` de `CEntier` n'est pas constante. Il faut ajouter le mot clef `const` après la déclaration de la fonction. Cet attribut est valide depuis la version 2.0 du C++.

```
int valeur() const { return a; }
```

Une méthode constante est un attribut dérivé. Il s'agit de l'implantation d'un attribut déduit des informations de l'objet, sans modification de celui-ci. L'appel d'une méthode d'un objet peut le modifier. Si une fonction reçoit un paramètre constant, elle indique qu'elle ne désire pas le modifier. Le compilateur ne peut pas savoir quelles méthodes modifient l'objet. Pour pouvoir différencier les deux types de méthodes, la syntaxe permet d'ajouter après la déclaration de celle-ci, le mot clef `const`, juste avant l'accolade ouvrante ou avant le point virgule. Avec cet attribut, la méthode considère le pointeur `this` comme pointant sur une constante. Plus précisément, pour une classe `CEntier`, une méthode constante reçoit le pointeur `this` du type : « `const CEntier* const this` ». Il y est alors impossible de modifier l'objet. Cela permet aux utilisateurs de l'objet de n'utiliser que les méthodes constantes s'ils ne désirent pas le modifier.

Il est à noter que Bjarne Stroustrup a déclaré le paramètre `this` en pointeur et non en référence, car à l'époque, les références n'étaient pas présentes dans le langage.

Une méthode constante est un attribut dérivé. En effet, ne modifiant pas l'objet, une méthode constante permet de retourner une information. Cette information est une vue partielle de l'objet. Elle est déduite de l'objet sans le modifier. L'objet peut tout aussi bien avoir un attribut pour cette information, ou effectuer un calcul complexe pour l'obtenir. Ce choix fait partie de l'implantation de la classe, pas de son usage. Par exemple, une classe `CHomme` peut posséder la date de naissance de l'individu. A partir de cette information, il est possible d'écrire une méthode constante retournant l'âge de celui-ci ou si l'homme en question est un adulte. Ces informations peuvent être déduites de la date de naissance :

```
class CHomme
{ long _dateNaissance;                      // en jour depuis 1/1/1900
  public:
  int getAge() const
  { return (JourCourrant-_dateNaissance); }
  bool isAdulte() const
  { return (getAge()>=18); }
};
```

ou au contraire, être présente en tant qu'attribut dans l'objet :

```
class CHomme
{ long _dateNaissance;
  bool _adulte;
  long _age;
  public:
  int getAge() const
  { return _age; }
  bool isAdulte() const
  { return _adulte; }
};
```

Toute méthode constante est une sorte d'attribut. Une méthode de calcul complexe recevant dix paramètres et en modifiant la moitié, si la méthode ne modifie pas l'objet, est un attribut dérivé.

Il peut arriver qu'une méthode soit par *principe* constante, mais que l'implantation désire malgré tout modifier l'objet auquel elle s'applique. C'est le cas lorsque l'on désire implanter un cache sur la méthode. Un attribut dérivé est déduit des attributs de l'objet, mais ce calcul peut être long. Il est intéressant d'optimiser cela en gardant le résultat du calcul dans un cache. Par exemple, une liste chaînée peut posséder une fonction comptant son nombre d'éléments. Celle-ci ne devrait pas modifier la liste, donc la méthode doit être constante. Il peut être utile de garder ensuite la taille de la liste dans une variable de l'objet pour la renvoyer lors des prochains appels. Il faut pour cela modifier l'objet. On utilise alors une conversion du pointeur `this` pour le rendre non-constant.

```
(CEntier*)this->nb=3;
```

Il n'est pas possible d'écrire :

```
{ CEntier* p=(CEntier*)this;
  p->nb=3;
}
```

pour avoir un accès plus facile à tous les attributs non-constants (Voir règle CPP.STY.7, page 228). Il faut utiliser une conversion à chaque utilisation des attributs.

Le comité ANSI/ISO a accepté une nouvelle syntaxe. Il est possible d'ajouter l'attribut `mutable` à l'objet `nb`. Cela permet de le modifier dans une méthode constante sans convertir le `this`.

D'autre part, si vous rédigez deux méthodes surchargées, l'une constante, et l'autre non, l'appel n'est pas forcément celui que l'on imagine.

```
class CEntier
{ int x;
  public:
    CEntier(int z) : x(z) {}
  int  valeur() const  { return x; }
  int& valeur()        { return x; }
};

void main()
{ CEntier a=3;
  int z;

  a.valeur()=2; // Appel de "CEntier::valeur()"
  z=a.valeur(); // Appel de "CEntier::valeur() const"
                // ou "CEntier::valeur()" ?
}
```

Si un objet est utilisé en *rvalue* (valeur à droite d'une expression d'assignation), l'appel s'effectue sur la méthode non constante ! Étant donné que l'on ne désire pas modifier l'objet `a`, on s'attend à ce que l'appel s'effectue sur la version constante de la méthode `valeur()`. Ce n'est pas le cas. Cette méthode n'est utilisée que lorsque l'objet `a` est lui-même constant.

D'autre part, les chaînes de caractères constantes sont implantées de différentes façons selon les options des compilateurs. Une chaîne constante est présente dans le programme au même titre qu'une variable globale. Il est possible de modifier cette chaîne. Une écriture comme cela :

```
*"abc"='d';
```

est valide en C ou C++. Le compilateur traduit cela en :

```
static char _str1[]={'a','b','c','\0'};
...
*_str1='d';
```

Il est possible de modifier la valeur d'une chaîne constante ! Ce n'est pas académique et c'est même à proscrire. Certains compilateurs offrent la possibilité de réunir les chaînes identiques dans le même tampon mémoire. Une option permet de réunir ces chaînes dans le même module. Sans cette option, le compilateur génère pour :

```
void main()
{ char buf[100];
  sprintf(buf,"%d",12);
  printf("%d",15);
}
```

un code proche de cela :

```
static char _str1[]={'%','d','\0'};
static char _str2[]={'%','d','\0'};

void main()
{ char buf[100];

  sprintf(buf,_str1,12);
  printf(_str2,15);
}
```

Avec l'option de détection de chaînes identiques cela donne :

```
static char _str1[]={'%','d','\0'};

void main()
{ char buf[100];

  sprintf(buf,_str1,12);
  printf(_str1,15);
}
```

Dans ce cas, la modification d'une chaîne de caractères constante, impacte toutes les versions de cette même chaîne. L'ajout au début du programme de `*"%d"='#'` modifie l'exécution des fonctions `sprintf` et `printf`.

La détection des chaînes identiques s'effectue sur la fin de celles-ci. Par exemple, le programme suivant :

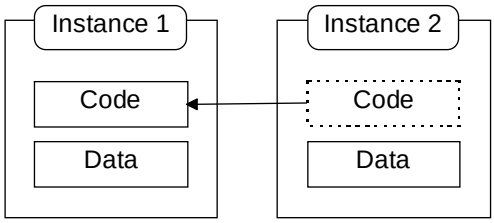
```
void main()
{ const char* nom1="frederic";
```

```
  const char* nom2="eric";
}
```

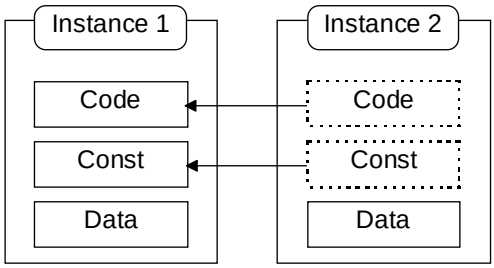
sera compilé comme cela :

```
static char str[]={'f','r','e','d','e','r','i','c','\0'};
void main()
{ const char* nom1=str;
  const char* nom2=str+4;
}
```

Pour empêcher les modifications d'une chaîne constante, certains compilateurs offrent une option supplémentaire qui permet de placer toutes ces chaînes de caractères dans un segment mémoire en lecture seule. Il n'est alors plus possible de les modifier. La détection des chaînes identiques ne pose plus de problème. Si un programme modifie une chaîne constante, une violation mémoire est générée et le programme s'arrête. Ce segment constant permet d'optimiser l'utilisation mémoire du programme lorsque plusieurs instances de celui-ci sont chargées par le système d'exploitation. En effet, si le même programme est chargé par le système plusieurs fois, seules les nouvelles données du programme sont ajoutées à celui-ci. Le programme n'étant pas modifié, c'est le même code qui est utilisé.



Le segment mémoire constant peut être partagé entre plusieurs instances d'un même programme.



Toutes les chaînes de caractères constantes sont partagées entre plusieurs instances du programme. Dans ce cas, le compilateur peut modifier le type des chaînes pour les déclarer constantes. Une chaîne `"abc"` est normalement du type `char*`, elle devient avec cet artifice du type `const char*`. Cela n'a généralement pas d'impact sur le programme. Le compilateur refuse simplement les écritures du type `*"abc"='C'`. Par contre, si vous déclarez incorrectement les paramètres de vos fonctions ou de vos méthodes, les appels de celles-ci risquent de ne plus fonctionner.

Prenons une fonction recevant un pointeur de caractères.

```
void f(char* p);
```

Si cette fonction ne modifie pas les caractères pointés par `p`, il n'est plus possible de l'appeler avec une constante « chaîne de caractère ». L'appel de `f("abc")` sera refusé par le compilateur. La conversion d'un pointeur constant vers un pointeur non constant est refusée, à juste titre, par le compilateur. Si vous modifiez la signature de votre fonction par

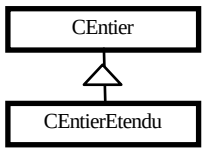
```
void f(const char* p);
```

vous pourrez l'appeler à l'aide d'une constante « chaîne de caractères ». De plus, le compilateur vérifiera que la fonction ne modifie réellement pas les caractères pointés. Une écriture comme `p[2]='a'` sera rejetée par le compilateur, et non à l'exécution lors de la violation du droit d'accès à la mémoire.

Toujours ajouter l'attribut `const` à tous les niveaux, dans les paramètres, les retours de fonctions, les références, et le type des méthodes.

Règle CPP.STY.7

```
class CEntier
{ protected:
  int x;
};

class CEntierEtendu : public CEntier
{
  void f()
  { CEntier* p=this;
    x=5;    // Ok
    p->x=3; // Erreur
  }
};
```



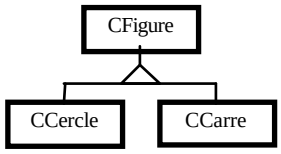
L'accès à `x` *via* `p` dans la fonction `CEntierEtendu::f()` est impossible. Pourquoi et comment corriger ?

L'accès à `x` est externe à l'objet `this`, donc les droits sont les mêmes qu'en dehors de l'objet. Pour pouvoir écrire cela, il faut ajouter dans la classe `A` la commande « `friend class CEntierEtendu` ». Les droits d'accès aux données ne sont valides que par héritage et sur l'objet courant. Pour modifier un autre objet que `this`, il faut posséder les droits à l'aide d'un `friend` approprié.

Utiliser `friend` pour obtenir de l'extérieur, les droits sur un objet plutôt que rendre `public` les éléments.

Règle CPP.STY.8

```
class CFigure
{ int a;
};

class CCercle : public CFigure
{ int b;
};

class CCarre : private CFigure
{ int c;
};

void main()
{ CCercle  cercle;
  CCarre   carre;
  CFigure* pFigure;

  pFigure=&cercle;
  pFigure=&carre; // Erreur
}
```



Ce n'est pas compilable, pourquoi ? Comment corriger ?

La conversion d'un pointeur sur un objet dérivé, vers un pointeur sur une classe de base, ne fonctionne que si l'héritage est public. Sinon, il faut forcer la conversion dans le source.

```
  pFigure=(CFigure*)&carre;
```

Ceci s'explique parce que l'héritage non public, fait d'un objet de type `CCarre` un objet qui ne doit pas être vu comme une sorte d'objet `CFigure`. Sinon, l'héritage public conviendrait. Pour éviter les conversions implicites, la norme indique que celles-ci sont refusées. Elle laisse quand même la possibilité de la forcer, mais dans ce cas, c'est de la responsabilité du programmeur d'ajouter une conversion.

Il est à noter que la future norme du C++ introduit un nouveau mot clef `explicit` pour interdire les conversions implicites de types.

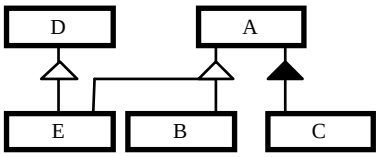
```
class A
{ public:
```

```
  explicit A(int i);
  explicit operator int();
};

void main()
{ long li=0;
  A a(li); // Erreur, conversion implicite
  li=a;    // Erreur
  a=3L;    // Erreur
};
```

Ne pas convertir en type de base, un pointeur d'un objet hérité de façon `protected`.

Règle CPP.STY.9

```
struct A             { int mya; };
struct B :         A { int myb; };
struct C : virtual A { int myc; };
struct D             { int myd; };
struct E : D, A      { int mye; };

void main()
{
  B b;
  C c;
  E e;
  A* ptA;

  ptA=&b;
  cout << ((void*)ptA==(void*)&b) << endl;

  ptA=&c;
  cout << ((void*)ptA==(void*)&c) << endl;

  ptA=&e;
  cout << ((void*)ptA==(void*)&e) << endl;
}
```



Qu'affiche `main` ?

`main` affiche `1` puis `0` et `0`. En effet, une conversion d'un objet vers sa première base ne change pas l'adresse de l'objet, par contre, une conversion vers une classe héritée virtuellement, ou sa deuxième base, modifie le pointeur (Voir « Héritages », page 265) !

Avec un héritage virtuel, une double conversion en aller-retour comme suit :

```
ptA=(C*)(A*)&c;
```

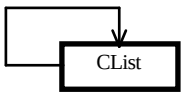
est refusée par le compilateur. Il n'est pas possible de partir d'une classe de base héritée virtuellement et d'arriver à une classe dérivée. Il est à noter que pour `(A*)(C*)NULL`, le pointeur n'est pas ajusté, le résultat reste `NULL`.

La future norme du C++ permettra les convertions inverses à l'exécution en utilisant un `template` particulier et le Run-Time-Type-Identification (Identification du type à l'exécution).

Éviter de convertir un objet en `void*`.

Règle CPP.STY.10

```
class CList
{ const CList& next;
  public:
    CList(const CList& x)
  : next(x) {}
};
```



Comment créer uniquement la *première* instance de `CList` ?

Pour avoir une référence sur un objet, il faut déjà avoir un objet de ce type ! Cela rappelle le paradoxe de la poule et de l'oeuf. Qui était avant ? Le premier constructeur attend qu'il existe déjà un autre objet de type `CList` ! Il faut faire croire qu'il en existe un, comme cela :

```
CList PremierCList(*(CList*)NULL);
```

Cela permet par la suite, d'avoir un code du genre « `if (&next==NULL)` ».

Ne pas déclarer de référence artificielle sur l'adresse `NULL`.

Règle CPP.STY.11

```
class CEntier
{ public:
   CEntier(int) {}
};

void f(CEntier& x)
{ // ...
}

void main()
```

```
{ f(3); // Erreur
}
```

Ce n'est pas compilable, pourquoi ?

Pour appeler la fonction `f()`, le compilateur doit obtenir une référence sur un objet de type `CEntier`. Pour cela, il doit créer un objet temporaire à l'aide du constructeur de `CEntier` recevant un entier comme paramètre. Ensuite, il peut envoyer la référence de cet objet temporaire. Dans la nouvelle norme du C++, il n'est plus possible de construire un objet temporaire pour obtenir une référence non constante. Cela entraînerait que la fonction `f()` modifierait l'objet temporaire avant qu'il ne soit détruit, ce qui n'a aucun sens. Cela entraînait une confusion chez les programmeurs. Maintenant, un objet temporaire n'est créé que pour obtenir une référence constante. Il faut modifier la déclaration de la fonction `f()` comme cela : « `void f(const CEntier& x)` ».

Toujours utiliser une référence constante si l'objet référencé n'est pas modifié.

Cela entraîne qu'il faut déclarer correctement les méthodes d'un objet pour indiquer au compilateur si elles modifient ou non l'objet courant. L'attribut `const` doit être ajouté afin de pouvoir utiliser l'objet à l'aide d'une référence constante.

Règle CPP.STY.12

CStatic

```
class CStatic
{ public:
  CStatic() { cout << "CStatic::CStatic()" << endl; }
};

void f()
{ cout << "f()" << endl;
  static CStatic StaticA;
}

void main()
{ cout << "main()" << endl;
  f();
  f();
}
```

Qu'affiche `main`, pourquoi ?

`main` affiche :

```
main()
f()
```

```
CStatic::CStatic()
f()
```

On pourrait penser que les objets statiques déclarés dans une fonction sont initialisés en même temps que ceux déclarés hors des fonctions. Ce n'est pas le cas. Le compilateur maintient un drapeau indiquant si l'initialisation a été effectuée. Cela peut avoir des effets de bord si le constructeur de l'objet a une influence sur d'autres objets statiques globaux. Ce choix a été fait pour initialiser les variables statiques avant leurs premières utilisations et non avant le `main`. Cela permet de contrôler le contexte d'initialisation. Sans cette technique, toutes les variables statiques des fonctions des librairies seraient initialisées, même si le programme ne les utilise jamais. Dans l'exemple suivant, l'appel récursif de la fonction n'est exécuté que lors du premier passage et avec le paramètre `i`. Ceci est impossible si l'objet statique est déclaré hors de la fonction.

```
int recursif(int i)
{ static int s=recursif(i-1);              // Executé la premiere fois
  return s;
}
```

Déclarer si c'est possible, les objets statiques dans la fonction qui les utilise plutôt qu'en dehors de celle-ci.

Règle CPP.STY.13

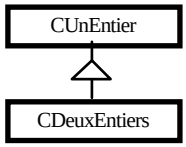
```
int z;
int& a=z;
int& const b=z;
```

Une référence ne peut être initialisée qu'à la construction. Elle se comporte donc comme une constante. L'opérateur égal d'une référence, modifie l'objet référencé, mais pas la référence elle-même. `b` est une référence constante et non une référence sur une constante. Quelle est la différence entre `a` et `b` ?

A priori il n'y a aucune différence. Sauf dans un cas : si ces écritures sont déclarées globales au fichier. Les objets constants et globaux au fichier ne sont pas déclarés `public`. Donc `a` est `public`, `b` ne l'est pas. Pour pouvoir utiliser `b` dans un autre fichier, il faut auparavant déclarer « `extern int& const b;` ».

Déclarer en `extern` les objets pouvant être utilisés par un autre fichier.

Règle CPP.STY.14




```
class CUnEntier
{ public:
  int a;
    CUnEntier() : a(0) { }
};


class CDeuxEntiers : public CUnEntier
{ public:
  int b;
    CDeuxEntiers() : b(1) { }
};

void f(CUnEntier* tab1)
{ tab1[1].a=2;
}

void main()
{ CDeuxEntiers tab2[2];

  f(tab2);
  cout << tab2[0].a << endl;
  cout << tab2[0].b << endl;
  cout << tab2[1].a << endl;
  cout << tab2[1].b << endl;
}
```

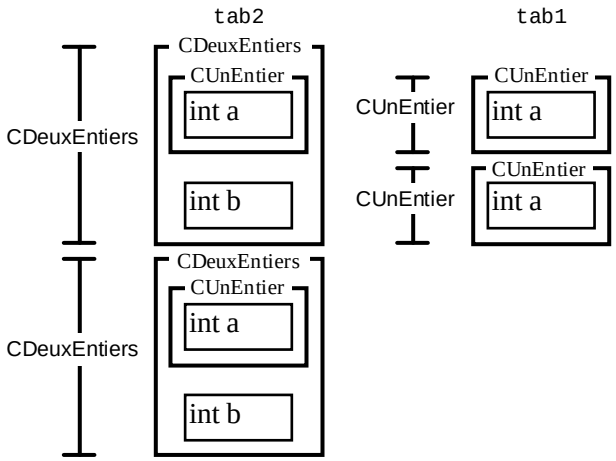
Qu'affiche `main`, pourquoi ?

`main` affiche :

```
0
2
0
1
```

Le compilateur ne sait pas faire la différence entre un pointeur et un tableau d'objets. Dans le cas présent, le tableau d'objets `CDeuxEntiers` est vu comme un tableau d'objets `CUnEntier`. L'adresse de l'élément 1 est calculée par rapport à `sizeof(CUnEntier)` et non *via* `sizeof(CDeuxEntiers)`. L'adresse pointée est donc erronée. L'élément `b` du premier élément du tableau `tab2` est modifié comme étant un objet de type `CUnEntier`.

La mémoire est vue comme cela :



Déclarer dans le prototype d'une fonction les crochets d'un paramètre tableau si la fonction ou la méthode utilise le pointeur comme un tableau.

Cela ne permet pas au compilateur de détecter les erreurs, mais c'est un moyen de décrire comment la fonction considère le pointeur.

Règle CPP.STY.22

```
class CEntier
{ int i;
  public:
    CEntier() : i(0) {}
  void print(ofstream& o)
  { o << "CEntier=" << i; }
};
```

Cette écriture limite l'utilisation de la classe. Pourquoi ?

Il n'est pas possible d'appeler la méthode `print` avec un flux `strstream`.

Il faut utiliser dans les méthodes les types les moins riches suivant l'utilisation qu'en fait la méthode. Dans l'exemple précédent, remplacer `ofstream` en `ostream` permet d'utiliser cette méthode avec tout type de flux, présent et futur. Si demain, un nouveau flux permet

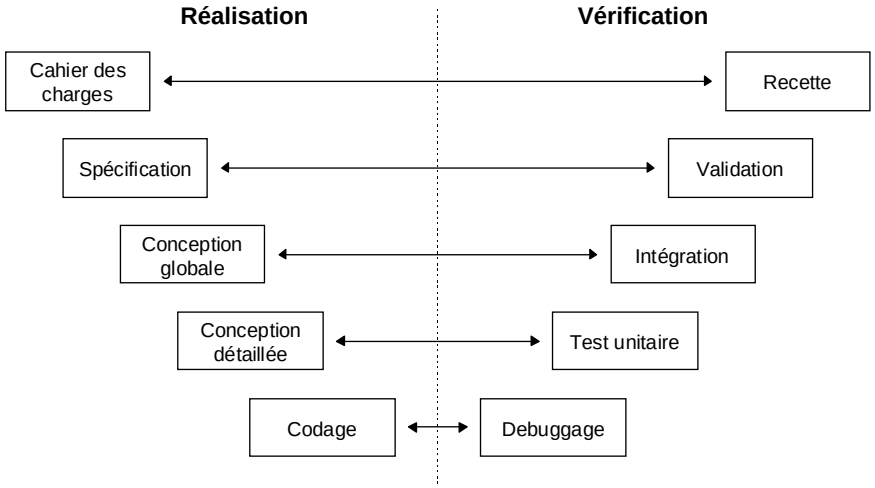
d'écrire dans une fenêtre, cette méthode sera toujours valide. Un des avantages du modèle objet réside dans cette utilisation. Il faut utiliser les classes les plus proches de la classe de base.

Utiliser le type de classe le moins riche dans la hiérarchie suivant l'utilisation qui en est faite.

CHAPITRE 8

TESTS

Pour valider un développement en C++, il faut utiliser l'approche traditionnelle. Le développement est découpé en plusieurs phases. Celles-ci sont de plus en plus précises. Chacune d'entre elles doit être validée.



Il n'est pas possible d'écrire du premier coup une classe importante sans que plusieurs erreurs apparaissent. Celles-ci peuvent venir d'une mauvaise utilisation du langage (d'où l'existence des règles spécifiques au C++ des chapitres précédents), ou d'une résolution erronée ou incomplète des spécifications de la classe.

Le débuggage est une phase difficile du C++. Il faut posséder de très bons outils pour pouvoir suivre précisément toutes les étapes cachées par le compilateur. Il n'est pas raisonnable d'ajouter des traces par-ci, par-là, pour vérifier les différents appels. La compilation est très consommatrice en temps CPU car le compilateur effectue énormément de travail. L'édition de lien est également très coûteuse. Recompiler successivement pour n'ajouter que quelques traces, n'est pas envisageable.

Certains outils peuvent être écrits facilement, d'autres devront être achetés. Un bon débuggeur, spécifique au C++, est nécessaire. Les débuggeurs ne permettent de détecter que les erreurs fatales. Les erreurs sur la valeur d'un attribut ou sur la sémantique d'une méthode ne peuvent pas être détectées par un débuggeur. Tout au plus, vous pouvez parcourir le code pour constater l'appel erroné d'un service. Avec l'aide de quelques outils très simples, vous pouvez détecter un nombre très important d'erreurs que le débuggeur ne trouvera pas, et cela, dès qu'elles arrivent.

A. INVARIANT, PRÉ ET POSTCONDITIONS

Chacune des méthodes d'une classe doit répondre à un contrat. Elles reçoivent des paramètres en entrées répondant à certaines contraintes, et elles doivent effectuer les traitements attendus par l'appelant. Pour cela, à la fin de celles-ci, il est envisageable de vérifier si les traitements se sont correctement effectués. Bertrand Meyer, le concepteur du langage objet « Eiffel », a introduit les trois principes suivants :

- préconditions, les conditions devant être remplies avant d'appeler une méthode,
- postconditions, les conditions devant être présentes à la fin d'une méthode,
- invariants, les conditions étant toujours vraies.

Ces tests devant toujours être vrais sont des assertions.

Les assertions servent :

- à la production de programme correct
- à enrichir la documentation
- sont une aide à la mise au point

L'appel d'une méthode est un contrat. Ce contrat peut par exemple se traduire comme cela :

	Obligations	Bénéfice
Programmeur du client	N'appeler la routine que si `x`>= 0, `epsilon` >=10^{-6}	Obtenir en retour l'approximation de la racine carrée.
Rédacteur de la méthode	Renvoyer l'approximation demandée	Inutile de traiter les cas ou `x`>=0, ou `epsilon`<=10^{-6}

Chacun des protagonistes doit respecter sa part du contrat. Les assertions sont écrites pour vérifier l'exécution de celui-ci. Il ne s'agit pas de vérifier lors de l'exécution les paramètres pour renvoyer un code d'erreur. Cela doit être codé classiquement. Une précondition permet justement d'éviter de gérer les cas d'erreurs. Les assertions ne sont utiles que dans la phase de développement, pas dans la phase de production. L'utilisateur final ne doit pas être pénalisé par les tests d'assertions.

Le langage Eiffel est le premier langage qui offre dans sa syntaxe la prise en compte de ces principes. Les vérifications de ces conditions ne sont nécessaires que lors de la phase de débuggage et d'intégration. Il faut pouvoir les supprimer lors des phases de validation et/ou de recette. En C++, il est facile de traduire cela. Le préprocesseur va nous aider.

Le C ANSI déclare une macro appelée `assert()` permettant de vérifier une condition devant être vraie. Si celle-ci n'est pas vérifiée, le programme est interrompu et un message d'erreur indique le fichier et la ligne où l'erreur s'est produite. Cette macro peut être écrite comme cela :

```
#define assert(TST)                                                 \
          ((TST) ? (void)0                                          \
                 : (cerr << __FILE__ "(" << __LINE__                \
                         << "): Assertion failed " #TST             \
                         << endl,abort()))
```

La norme C++ demande à ce que `assert()` soit une expression. L'utilisation de celle-ci est extrêmement simple.

```
void f(char* pt)
{ assert(pt!=NULL);                          // abort() si pt==NULL
  //...
}
```

Un `#define` particulier permet de supprimer l'ensemble des appels à `assert()` lors de la compilation avant la recette. Si vous déclarez `#define NDEBUG`, tous les `assert` sont supprimés.

```
#ifndef NDEBUG
#define assert(TST) ...
#else
#define assert(TST) ((void)0)
#endif
```

Si vous désirez effectuer un traitement particulier pour vérifier les conditions d'un traitement, encadrer celui-ci par :

```
#ifndef NDEBUG
// Traitement spécial de vérification du programme
#endif
```

Nous allons utiliser cet outil pour écrire des macros spécifiques de vérification des invariants, des pré et postconditions.

```
#include <assert.h>
#define INVARIANT(TST)     assert(TST)
#define PRECONDITION(TST)  assert(TST)
#define POSTCONDITION(TST) assert(TST)
```

Une version étendue peut recevoir un message explicatif supplémentaire.

```
#define ASSERTMSG(TST,MSG)                                          \
          ((TST) ? (void)0                                          \
                 : (cerr << __FILE__ "(" << __LINE__                \
                             << "): Assertion failed " #TST         \
                             << MSG << endl,abort()))
```

On constate l'avantage du C++ sur le C. En effet, il est possible d'écrire :

```
ASSERTMSG(i>100,"Valeur trop petites");
ASSERTMSG(pt!=NULL,"this=" << *this);
```

Si les opérateurs de flux ont été écrits pour chaque objet, il est très facile d'afficher l'état de ceux-ci. De plus si vous utilisez les indentations de flux (Page 123), cette trace est facile à lire.

Les macros de tests deviennent :

```
#define INVARIANT(TST,MSG)     ASSERTMSG(TST,"Invariant " << MSG)
#define PRECONDITION(TST,MSG)  ASSERTMSG(TST,"Precondition " << MSG)
#define POSTCONDITION(TST,MSG) ASSERTMSG(TST,"Postcondition " << MSG)
```

Après la phase de débuggage, les « Postconditions » et les « Invariants » peuvent être considérés comme corrects. Il suffit alors d'adapter les macros précédentes, pour ne sup-

primer de la compilation que ces deux tests. Les « Préconditions » sont très importantes lors de la phase d'intégration. En effet, l'utilisation erronée de votre classe par les autres modules sera détectée à l'aide des préconditions. Le programme se testera de lui-même lors de son exécution. Qui mieux que lui peut le faire ?

Comment intégrer cela dans une classe ? Nous allons prendre un exemple simple. Prenons une classe `CAdulte` ayant différents attributs.

```
class CAdulte
{ int  _age;
  char _nom[40];
  char _prenom[40];
  public:
    CAdulte(const char* nom,const char* prenom,int age)
  : _age(age)
  { strcpy(_nom,nom);
    strcpy(_prenom,prenom);
  }
};
```

Un adulte doit toujours avoir son âge compris entre 18 et 150 ans. Pour le moment, il n'existe pas de personne ayant vécu plus de 150 ans. L'invariant de la classe `CAdulte` doit être écrit dans une méthode particulière. Celle-ci sera vide lors de la phase de recette.

```
#ifndef NDEBUG
void invariant() const
{ INVARIANT((_age>=18) && (_age<=150),"Age incorrect");
}
#else
void invariant() const {}
#endif
```

Le constructeur de `CAdulte` reçoit différents paramètres. Il est possible de vérifier ceux-ci en précondition.

```
CAdulte(const char* nom,const char* prenom,int age)
: _age(age)
{ PRECONDITION(strlen(nom)<sizeof(_nom)-1,"Nom trop grand");
  PRECONDITION(strlen(prenom)<sizeof(_prenom)-1,"Prenom trop grand");
  strcpy(_nom,nom);
  strcpy(_prenom,prenom);
  POSTCONDITION(!strcmp(_nom,nom),"Erreur sur le nom");
  POSTCONDITION(!strcmp(_prenom,prenom),"Erreur sur le prenom");
  invariant();
}
```

A la fin du constructeur, les invariants de la classe doivent être résolus. Chaque méthode autre que le constructeur, peut commencer par appeler la méthode `invariant()` pour vérifier l'état courant de la classe. Il est préférable de découvrir les erreurs le plus tôt

possible. Bien sûr, cela ralentit le programme. Tout dépend de la complexité de la méthode `invariant()`. Si par exemple, pour une relation `0..n` ↔ `0..n` vous testez en `invariant()` si tous les objets en relations possèdent bien la relation inverse, cela est très consommateur en temps CPU. Dans ce cas, vous pouvez choisir de ne vérifier les invariants que dans les méthodes non constantes. Les méthodes constantes ne devant pas modifier l'objet, il ne devrait pas y avoir d'effet de bord.

Le destructeur doit vérifier avant toute destruction les invariants, c'est un passage obligé pour les utilisateurs de la classe, donc le lieu idéal pour vérifier la classe.

Vous pouvez également rendre `public` la méthode `invariant()` ce qui permet aux appelants de vérifier l'objet. Une méthode `invariant()` peut ainsi vérifier l'état de son objet, et l'état de tous les objets en relation. Le test d'invariant peut être propagé dans tous les objets. La validation de l'invariant d'un objet entraîne le test des invariants de tous les attributs de l'objet et de tous les objets en relation. Au final, il existe un invariant de l'application elle-même qui consiste à vérifier les invariants de tous les objets de l'application.

Il est difficile de vérifier toutes les préconditions dans les constructeurs. En effet, ceux-ci commencent par appeler les constructeurs des classes de bases, et les constructeurs de leurs attributs. Les préconditions ne peuvent être rédigées que dans les accolades. Certains traitements sont alors déjà effectués, peut-être de façon erronée. Le constructeur ne pourra que constater, après coup, l'exécution erronée.

```
class A
{ B _b;
  public:
    A(int b) : _b(b)
  { PRECONDITION(b>=0,"b positif");         // B::B() déja appelé !
  }
  //...
};
```

Le test de précondition est effectué après la construction de l'attribut `_b`. Celui-ci n'aurait jamais dû être construit. Mais, si un entier négatif est valide pour cet objet, le constructeur de `_b` ne réagira pas. Sinon, la précondition de `B::B(int)` aurait refusé la construction. La précondition du constructeur de `A` sera testée un peu trop tard mais sera quand même détectée. Dans de très rares cas, il est possible qu'un traitement erroné soit exécuté avant le test de la précondition.

Pour vérifier une postcondition, il est parfois nécessaire de comparer le résultat d'une méthode avec l'état de l'instance avant celle-ci. Ce qui peut se traduire en C++ comme suit :

```
void A::f()
{
  PRECONDITION(...)
```

```
  #ifndef NDEBUG
  A old(*this); // Constructeur de copie
  #endif
  // ... Traitement
  POSTCONDITION(x==old.x * 2,"Calcul errone");
}
```

Il est parfois difficile de vérifier les postconditions des méthodes si celles-ci retournent des objets.

```
A B::getA()
{ return f();                              // cctr
}
```

Comment tester que la fonction `f()` retourne un objet répondant aux postconditions ? Il n'est pas correct de garder le résultat de `f()` dans une variable de la méthode pour ensuite la renvoyer, car cela n'est pas efficace.

```
A B::getA() const
{ A tmp=f();                               // cctr
  POSTCONDITION(A==1,"A doit être egal à 1");
  return tmp;                              // cctr
}
```

Avec ce type d'écriture, il y a deux appels au constructeur de copie à la place d'un seul. Une écriture alternative serait celle-là :

```
A B::getA() const
{
  #ifndef NDEBUG
  A tmp=f();
  POSTCONDITION(A==1,"A doit être egal à 1");
  return tmp;
  #else
  return f();
  #endif
}
```

L'inconvénient de cette approche est que le code est découpé en deux versions distinctes. Il n'est pas utile de vérifier la postcondition d'une version ne devant pas être exécutée en phase d'intégration. Les assertions doivent vérifier le vrai code. Il est parfaitement envisageable, que la version vérifiant la postcondition soit correcte, mais pas la version sans le test de postcondition. Seul un langage spécialisé peut vérifier au bon moment les assertions. Avec le C++ actuel, vous pouvez tester la plupart des assertions, mais pas toutes.

Lors de la rédaction des postconditions, il n'est pas nécessaire de vérifier les traitements simples. Par exemple, il n'est pas utile de vérifier qu'une variable à bien la valeur que l'on vient d'y mettre. Il faut avoir un minimum de confiance dans le compilateur. Seuls les traitements complexes doivent être vérifiés. Par exemple, pour optimiser une méthode, il est utile de rédiger une version simple de l'algorithme qui sera facilement vérifiable mais lente. Cette version servira d'étalon à la version optimisée. En postcondition, il est important de vérifier que la routine obtient le même résultat que la version non optimisée. L'optimisation consiste à détecter des cas particuliers pour améliorer les traitements ou à modifier l'algorithme de traitement pour éviter les redondances. Dans les deux cas, il peut y avoir des fuites sur des combinaisons particulières. Certaines situations peuvent être dirigées vers un traitement par erreurs. La réduction de la redondance peut supprimer par erreur une redondance justifiée. Pour détecter ces situations, il faut comparer le résultat avec la version simple de l'algorithme. Pour écrire une version optimisée, il faut dans un premier temps écrire une version lente mais simple. Une fois la version lente écrite, le programme peut fonctionner sans rédiger la version rapide. Un utilitaire peut alors détecter les routines consommant beaucoup de temps. Vous devrez alors ne modifier que les routines critiques ayant été identifiées. Commencez toujours par écrire une version simple de l'algorithme. Au pire, elle servira à valider la version rapide.

Lors de la phase de debuggage, vous devez déclarer :

	Au début	A la fin
Constructeur	Précondition	Invariant Postcondition
Méthode `const`	(Invariant) Précondition	(Invariant) Postcondition
Méthode non `const`	Invariant Précondition	Invariant Postcondition
Destructeur	Invariant Précondition	Postcondition

Une variante consiste à supprimer les Invariants des méthodes `const` car elles ne modifient pas l'objet. En phase d'intégration, vous pouvez ne laisser que :

	Au début	A la fin
Constructeur	Précondition	
Méthode `const`	Précondition	
Méthode non `const`	Précondition	

Destructeur	Précondition

car le corps de la classe est considéré comme validé par le test unitaire. Si vous êtes pessimiste, continuez à utiliser tous les tests présents lors de la phase de débuggage. Lors de la phase de validation ou de recette, tous les tests des assertions doivent être supprimés. Il faut recompiler tout le programme avec l'option `/DNDEBUG`.

Lors de la rédaction d'une assertion, il ne faut pas appeler de méthodes non `const` de la classe, car il ne doit pas y avoir d'effet de bord. De même, vous ne devez pas utiliser les mêmes variables que le corps de la classe. Encadrez le code de vérification d'accolade.

```
#ifndef NDEBUG
{
  //...
}
#endif
```

Le langage Eiffel utilise un environnement extérieur à la fonction pour décrire les assertions. En C++, il faut un minimum de rigueur pour obtenir le même effet. Il ne faut pas qu'en enlevant le code d'une assertion, le programme ait un comportement erroné. Les assertions ne sont pas là pour corriger le code, mais pour le vérifier. Les critères d'optimisations ou de places mémoires ne doivent pas impacter la rédaction de ce code, car l'application finale ne vérifiera plus les assertions. Les assertions ne doivent qu'ajouter du code, mais pas le modifier.

Il est tentant, lors de la rédaction des postconditions, de vouloir utiliser une autre méthode de la classe afin de confirmer le traitement par réversibilité. Par exemple, pour vérifier l'opérateur plus-égal, il semble intéressant de vérifier en postcondition, si en appelant l'opérateur moins-égal sur le résultat on obtient bien la valeur avant le traitement.

```
CFraction& operator +=(const CFraction& x)
{
  #ifndef NDEBUG
  CFraction old=*this;
  #endif
  //... Traitement
  #ifndef NDEBUG
  CFraction tmp=*this;
  POSTCONDITION((tmp-=x)==old,"Operateur += ou -= faux");
  #endif
  return *this;
}
```

Mais que va faire l'opérateur moins-égal ? De même, il va appeler l'opérateur plus-égal pour vérifier son comportement. Nous sommes en présence d'une récursivité infinie. L'opé-

rateur plus-égal appelle l'opérateur moins-égal, qui appelle l'opérateur plus-égal, qui appelle etc.

Il ne faut pas vérifier la réversibilité d'une méthode en postcondition. Ce sera fait en test unitaire. Par contre, certaines méthodes peuvent judicieusement appeler d'autres méthodes pour vérifier les postconditions. Ces méthodes doivent être constantes. Le constructeur de copie et l'opérateur d'affectation peuvent vérifier leurs comportements à l'aide de l'opérateur de comparaison d'égalité.

```
CFraction(const CFraction& x)
{ //... Traitement
  POSTCONDITION(*this==x,"Constructeur de copie faux");
}
CFraction& operator =(const CFraction& x)
{ //... Traitement
  POSTCONDITION(*this==x,"Constructeur de copie faux");
  return *this;
}
```

Les postconditions doivent être rédigées avec rigueur, elles doivent être valides quelles que soient les conditions d'appel. Pour les tests aux limites ou les tests de réversibilité, utilisez les tests unitaires. Peut-être qu'à l'avenir, le C++ possédera une extension gérant cela. Sara Porat et Paul Fertig ont fait une proposition dans ce sens (*cf.* « *Journal of Object-oriented programming* », Mai 1995).

B. TEST UNITAIRE

Le test unitaire est la première étape de validation d'un développement. Il ne faut pas confondre le test unitaire et le déverminage. Le test unitaire cherche à prouver qu'une classe est correcte, le déverminage permet de localiser une erreur. Le test unitaire est un complément des préconditions et des invariants (Voir page 238).

Les tests unitaires sont des exemples d'utilisation correcte de la classe. Ils vérifient que la classe fonctionne si on l'utilise correctement. Pour des scénarios d'utilisation « valide », si une précondition, une postcondition ou un invariant échoue, la classe et/ou le test sont erronés. Un test unitaire vérifiera qu'un appel à une méthode avec un jeu de paramètres connus donne le résultat attendu. Dans les tests unitaires, il faut également tester les méthodes aux limites. Les assertions sont un bon moyen de trouver les erreurs dans les tests unitaires.

Les tests unitaires permettent de vérifier la non-régression d'un développement. Avant chaque nouvelle intégration d'une classe, les tests unitaires doivent être refaits pour vérifier que la nouvelle version ne régresse pas par rapport à la précédente.

Quand concevoir les tests-unitaires ? Dès la phase de conception. En effet, la rédaction des tests unitaires fait apparaître des erreurs possibles qui pourront influencer la modélisation. Plus tôt les erreurs sont trouvées, moins l'impact est important. La correction de certaines erreurs peut entraîner une modification importante de l'interface ou de la modélisation d'une classe.

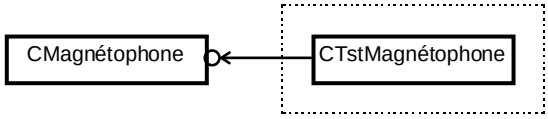
« Si vous n'avez pas la patience de tester votre programme, celui-ci testera votre patience ! »

Pour le modèle objet, il semble naturel d'effectuer les tests unitaires sur les classes. Il faut tester toutes les méthodes d'une classe et les transitions d'état de celle-ci. Nous allons prendre un exemple de classe dont nous voulons vérifier la conformité avec ses spécifications. Testons la classe `CMagnetophone`. Celle-ci possède un état lui permettant d'autoriser certaines méthodes.

```
class CMagnetophone
{ public:
  enum TEtat { STOP,START,REWIND };
    CMagnetophone()
  : _etat(STOP) { }
  virtual ~CMagnetophone()
  { }

  virtual void start()
  { PRECONDITION(_cassette==true,"Pas de cassette");
    PRECONDITION(_etat==STOP,"Deja en marche");
    _etat=START;
  }
  virtual void stop()
  { _etat=STOP;
  }
  int compteur() const;
  void ajouteCassette()
  { PRECONDITION(_cassette==false,"Deja une cassette");
    _cassette=true;
  }
  void enleveCassette()
  { PRECONDITION(_cassette==true,"Il n'y a pas de cassette");
    _cassette=false;
  }
  bool cassette() const
  { return _cassette;
  }
  protected:
  bool finBande() const;
  TEtat _etat;
  bool  _cassette;
};
```

a. Méthodes public

Dans un premier temps, nous allons tester les méthodes publiques de la classe. Pour cela, nous allons écrire une classe `CTstMagnetophone` s'occupant de tester ces méthodes. Cette classe maintiendra l'état courant du test, et l'état de tous les objets testés. Pour pouvoir appeler une méthode, il faut dans un premier temps créer un objet. La classe `CTstMagnetophone` gardera l'information de la création de la classe. Ainsi, le test d'une méthode pourra vérifier auparavant que l'instance est bien présente.




```
class CTstMagnetophone
{ CMagnetophone* _pMagneto;
  enum { IS_STOP,IS_START } _etat;
  public:
    CTstMagnetophone() : _pMagneto(NULL) { }
};
```

L'attribut `_etat` de `CTstMagnetophone` garde la trace de l'état courant de la classe `CMagnetophone`. Il n'est pas toujours possible de connaître l'état réel d'une classe. Il faut dans ce cas, maintenir un état hypothétique de la classe testée. Cela permet aux méthodes de vérifier les conditions de test avant de se lancer. Ensuite, nous allons ajouter des méthodes renvoyant trois types de valeur :

- test OK
- test erroné
- et condition de test incorrecte.

Pour effectuer un test, il faut que certaines conditions soient réunies. Chaque méthode de test vérifiera les conditions courantes du test et effectuera les appels aux méthodes de l'objet. Elles vérifieront les réponses de celles-ci suivant les valeurs attendues en sortie par rapport aux valeurs d'entrée. Les transitions nécessaires sur l'objet `CMagnetophone` seront également vérifiées. La plupart des méthodes de tests renverront par défaut la valeur `Ok` car les vérifications auront été faites dans les postconditions des méthodes testées. Parfois, une postcondition ne peut pas tester la validation d'une méthode car le résultat dépend d'un contexte qu'elle ne maîtrise pas. Dans ce cas, la méthode de test vérifiera le résultat attendu.

```
enum CTstRet { Ok,Bad,TstCondition };
class CTstMagnetophone
```

```
{ CMagnetophone* _pMagneto;
  enum { IS_STOP,IS_START } _etat;
  public:
    CTstMagnetophone()
  : _pMagneto(NULL), _etat(IS_STOP) { }

  // Test de la construction
  virtual CTstRet ctr1()
  { if (_pMagneto!=NULL) return TstCondition;
    _pMagneto=new CMagnetophone();
    if (_pMagneto==NULL) return Bad;
    if (_pMagneto->cassette()) return Bad;
    _etat=IS_STOP;
    return Ok;
  }



  // Test de la destruction
  virtual CTstRet dtr1()
  { if (_pMagneto==NULL) return TstCondition;
    delete _pMagneto;
    _pMagneto=NULL;
    return Ok;
  }

  // Test méthode Start dans n'importe quel etat
  virtual CTstRet start1()
  { if (_pMagneto==NULL)        return TstCondition;
    if (!_pMagneto->cassette()) return TstCondition;
    if (_etat!=IS_STOP)         return TstCondition;
    _pMagneto->start();
    _etat=IS_START;
    return Ok;
  }

  // Test méthode Stop dans n'importe quel etat
  virtual CTstRet stop1()
  { if (_pMagneto==NULL) return TstCondition;
    _pMagneto->stop();
    _etat=IS_STOP;
    return Ok;
  }

  // Test méthode Stop dans l'état START
  virtual CTstRet stop2()
  { if (_pMagneto==NULL) return TstCondition;
    if (_etat!=IS_START) return TstCondition;
    _pMagneto->stop();                      // Stop si etat START
    _etat=IS_STOP;
    return Ok;
  }

  // Test méthode ajouteCassette
  virtual CTstRet ajouteCassette1()
```

```
  { if (_pMagneto==NULL)       return TstCondition;
    if (_pMagneto->cassette()) return TstCondition;
    _pMagneto->ajouteCassette();
    if (_pMagneto->cassette()==false) return Bad;
    return Ok;
  }

  // Test méthode enleveCassette
  virtual CTstRet enleveCassette1()
  { if (_pMagneto==NULL)        return TstCondition;
    if (!_pMagneto->cassette()) return TstCondition;
    _pMagneto->enleveCassette();
    if (_pMagneto->cassette())  return Bad;
    return Ok;
  }
};
```

Toutes les méthodes de test vérifient le contexte courant du test. Elles effectuent le test et vérifient le comportement des méthodes de `CMagnetophone`. Il n'est pas toujours possible de vérifier le comportement d'une méthode à l'extérieur de la classe. Les postconditions s'occuperont de cela. Le test unitaire est un exemple d'utilisation de la classe. La méthode `start1()` appelle la méthode correspondante de la classe `CMagnetophone`, mais est incapable de vérifier si le comportement de la méthode est correct. Par contre, la méthode `ajouteCassette1()` vérifie le comportement de la méthode.

Il est possible, par la suite, d'écrire des scénarios de test de la classe `CMagnetophone`. Pour cela, il suffit d'appeler successivement différentes méthodes de tests. Les scénarios peuvent eux-mêmes être des tests de la classe `CMagnetophone`.

```
class CTstMagnetophone
{ //...
  virtual CTstRet scenario1()
  { if (_pMagneto!=NULL) return TstCondition;
    CTstRet rc;
    if ((rc=ctr1())!=Ok)   return rc;
    if ((rc=ajouteCassette1())!=Ok) return rc;
    if ((rc=start1())!=Ok) return rc;
    if ((rc=stop2())!=Ok)  return rc;
    if ((rc=start1())!=Ok) return rc;
    if ((rc=stop1())!=Ok)  return rc;
    rc=dtr1();
    return rc;
  }
};
```

Une autre façon d'écrire ce scénario en utilisant les pointeurs de membres est la suivante :

```
class CTstMagnetophone
{ //...
  virtual CTstRet scenario1()
  { static CTstRet (CTstMagnetophone::* tab[])()=
```

```
    {&CTstMagnetophone::ctr1,  &CTstMagnetophone::start1,
     &CTstMagnetophone::ajouteCassette1,
     &CTstMagnetophone::stop2,
     &CTstMagnetophone::start1,&CTstMagnetophone::dtr1
    };
    if (_pMagneto!=NULL) return TstCondition;
    CTstRet rc;
    for (int i=0;i<sizeof(tab)/sizeof(tab[0]);++i)
    { if ((rc=(this->*tab[i])())!=Ok) return rc;
    }
    return Ok;
  }
};
```

Il est également envisageable d'effectuer des tests aléatoires en s'appuyant sur l'état `TstCondition` des méthodes. Un tirage aléatoire choisit un test ou un scénario au hasard, puis celui-ci est exécuté. Si le test retourne l'état `TstCondition`, celui-ci est considéré comme n'ayant pas pu être fait, et un nouveau tirage est effectué. Une boucle de durée ou d'itération finie peut tester la classe dans les contextes les plus variés.

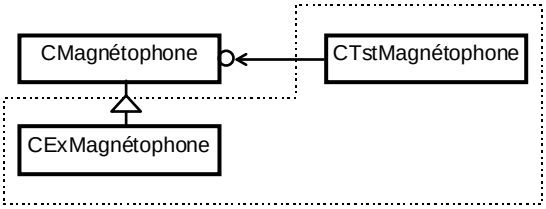
```
class CTstMagnetophone
{ //...
  virtual CTstRet scenario2()
  { static CTstRet (CTstMagnetophone::* tab[])()=
    {&CTstMagnetophone::ctr1,&CTstMagnetophone::dtr1,
     &CTstMagnetophone::ajouteCassette1,
     &CTstMagnetophone::start1,
     &CTstMagnetophone::stop1,&CTstMagnetophone::stop2
    };
    CTstRet rc;
    for (int i=0;i<50;++i)
    { do
      { rc=(this->*tab[rand() % (sizeof(tab)/sizeof(tab[0]))])();
      } while (rc==TstCondition);
      if (rc==Bad) return Bad;
    }
    return Ok;
  }
};
```

Il est également possible d'envisager un parcours exhaustif de toutes les transitions des états d'une classe, en rédigeant un test pour chaque changement d'état. Un automate parcourt ensuite l'ensemble des transitions possibles en appelant successivement les tests correspondants.

b. Méthodes `protected`

Cela permet de tester les méthodes `public` d'une classe. Il faut également tester les méthodes protégées de celles-ci. Pour cela, si l'on ne désire pas modifier la classe testée pour

effectuer le test (En déclarant toutes les méthodes publiques par exemple), il faut construire une nouvelle classe qui héritera de la classe à tester. Celle-ci rendra toutes les méthodes protégées de la classe en `public`.




```
class CExMagnetophone : public CMagnetophone
{ public:
  bool finBande() const
  { return CMagnetophone::finBande(); }
};
```

Le test de la classe devra instancier une classe `CExMagnetophone` à la place de `CMagnetophone`, mais pourra alors, tester les méthodes `protected`. Il peut être préférable de séparer les tests des méthodes `protected` des tests des méthodes `public`. Dans ce cas, ajouter un mode dans la classe `CTstMagnetophe` et vérifier dans les conditions de test de chacun si ce mode correspond au type courant. Par exemple, si ce mode est à `TstProtected`, les méthodes `public` et `protected` peuvent être testées. Par contre, si ce mode est à `TstPublic`, seuls les tests des méthodes `public`s seront testés.

c. Méthodes `private`

Il n'est pas possible d'avoir un accès aux méthodes `private` d'une classe. Pour cela, il faudrait modifier la classe `CMagnetophone` pour déclarer la classe `CTstMagnetophone` en `friend`.

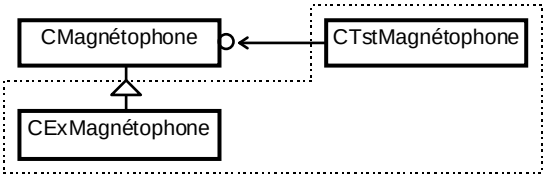
```
class CMagnetophone
{ ...
  friend class CTstMagnetophone;
  ...
};
```

Avec cette modification minime, il n'est plus nécessaire de construire la classe `CExMagnetophone`, car les méthodes `protected` et `private` deviennent accessibles au test. Il peut être raisonnable de ne pas tester les méthodes `private` car celles-ci ne sont utilisables que par les autres méthodes de la classe. Elles ne font pas partie de

l'interface de la classe et seront de toute façon testées par effet de bord, en appelant les méthodes `public` et `protected`. Il est très probable que ces méthodes évoluent par la suite sans remettre en cause l'utilisation de la classe. Les tests de ces méthodes deviendraient obsolètes. Seul l'interface de la classe doit être testée. Si la classe évolue, les tests de l'interface doivent toujours être valides.

d. Méthodes `virtual`

Pour tester les méthodes virtuelles, il faut vérifier leurs comportements si une classe dérivée modifie celles-ci. Pour cela, il faut déclarer cette classe dérivée et redéfinir toutes les méthodes virtuelles. C'est l'occasion d'offrir l'accès aux méthodes `protected`. Les méthodes virtuelles peuvent individuellement ou collectivement changer leurs comportements selon l'état de cette nouvelle classe.




```
class CExMagnetophone : public CMagnetophone
{ public:
  enum {Normal,Modif} _etat;

    CExMagnetophone()
  : _etat(Normal)
  { }

  void chgMode()
  { _etat=((_etat==Normal) ? Modif : Normal); }

  // Acces public pour finBande()
  bool finBande() const
  { return CMagnetophone::finBande(); }

  virtual void start()
  { if (_etat==Normal) CMagnetophone::start();
    else
    { // Modification du comportement
    }
  }

  virtual void stop()
  { if (_etat==Normal) CMagnetophone::stop();
    else
    { // Modification du comportement
```

```
    }
  }
};
```

La méthode `chgMode()` permet de basculer l'objet d'une version normale vers une version avec les méthodes virtuelles modifiées. Les tests unitaires des méthodes de la classe doivent continuer à être valides avec ou sans modification. Il est pertinent de tester la classe `CMagnetophone` seule, si elle n'est pas virtuelle pure, puis de parcourir les mêmes tests avec la classe `CExMagnetophone`.

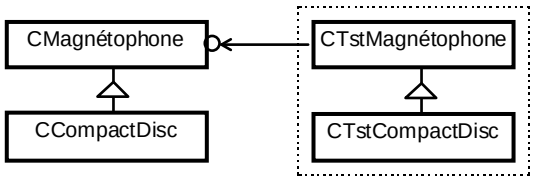
```
class CTstMagnetophone
{ CMagnetophone* _pMagneto;
  enum { IS_NOT,IS_STOP,IS_START,IS_REWIND } _etat;
  enum {IS_NORMAL,IS_VIRTUAL} _vir;
  public:
    CTstMagnetophone()
  : _pMagneto(NULL), _etat(IS_NOT), _vir(IS_NORMAL) { }

  // Construction de CMagnetophone
  virtual CTstRet ctr1()
  { if (_pMagneto!=NULL) return TstCondition;
    _pMagneto=new CMagnetophone();
    _vir=IS_NORMAL;
    if (_pMagneto==NULL) return Bad;
    _etat=IS_STOP;
    return Ok;
  }

  // Construction de CExMagnetophone etat Normal
  virtual CTstRet ctr2()
  { if (_pMagneto!=NULL) return TstCondition;
    _pMagneto=new CExMagnetophone();
    if (_pMagneto==NULL) return Bad;
    _vir=IS_NORMAL;
    _etat=IS_STOP;
    return Ok;
  }

  // Construction de CExMagnetophone etat Virtuel
  virtual CTstRet ctr3()
  { if (_pMagneto!=NULL) return TstCondition;
    _pMagneto=new CExMagnetophone();
    if (_pMagneto==NULL) return Bad;
    ((CExMagnetophone*)_pMagneto)->chgMode();
    _vir=IS_VIRTUAL;
    _etat=IS_STOP;
    return Ok;
  }
  //...
};
```

Les trois tests de construction vérifient le comportement de la classe dans différents cas d'utilisation. Le test `ctr2()` vérifie que la classe `CExMagnetophone` n'a pas ajouté d'erreur lors de l'utilisation normale. Le comportement de la classe doit être strictement le même que la classe originale. Le test `ctr3()` construit un objet `CExMagnetophone` en demandant la modification des méthodes virtuelles. Celles-ci peuvent modifier l'automate d'état. Dans ce cas, les tests doivent être adaptés pour tenir compte de ces changements.

Les classes abstraites peuvent également être testées par ce mécanisme. Les méthodes virtuelles pures seront redéfinies dans la classe `CExMagnetophone`.

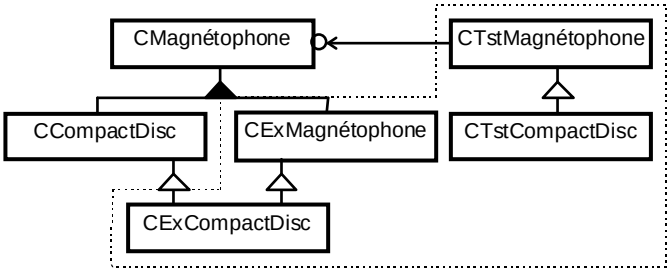
e. Héritage

Si une classe `CCompactDisc` hérite de `CMagnetophone`, celle-ci doit également être validée par un test unitaire. Elle doit être capable de répondre correctement à une majorité des tests de la classe `CMagnetophone`. Nous allons construire une classe `CTstCompactDisc` qui hérite de la classe `CTstMagnetophone`. Les tests supplémentaires peuvent alors être ajoutés pour les nouvelles méthodes de `CCompactDisc`, et les tests hérités peuvent être inclus dans les nouveaux scénarios.

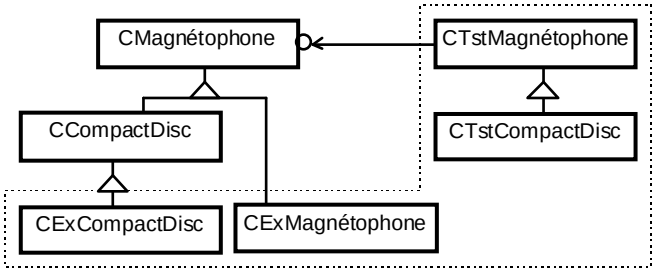


Certains tests de `CTstMagnetophone` peuvent être adaptés dans la classe `CTstCompactDisc` en redéfinissant les méthodes virtuelles.

On aimerait pouvoir recycler `CExMagnetophone` pour vérifier les méthodes `protected` de `CCompactDisc` et y ajouter les tests des nouvelles méthodes virtuelles. Pour cela, il faut que les classes `CExMagnetophone` et `CCompactDisc` héritent virtuellement de `CMagnetophone.` Cela donnerait le modèle suivant :



Malheureusement, la classe `CCompactDisc` n'hérite pas virtuellement de `CMagnetophone`. Il faut alors réécrire les méthodes de `CExCompactDisc` à l'aide, éventuellement, d'un copier-coller pour avoir ce modèle :



Les mêmes principes, suite aux différents types de méthodes, sont à utiliser pour tester correctement la classe `CCompactDisc`. Au fur et à mesure de l'enrichissement du modèle, les classes héritées bénéficieront des tests des classes de base. Si par la suite la classe `CMagnetophone` évolue, les tests correspondants seront modifiés. La classe `CTstCompactDisc` bénéficiera de ces modifications. Il faudra par contre, vérifier la classe `CExCompactDisc` pour vérifier l'ensemble des méthodes virtuelles déclarées. Il est possible que de nouvelles méthodes soient ajoutées dans la classe `CMagnetophone`. Les classes `CExMagnetophone` et `CExCompactDisc` doivent alors être adaptées.

f. Comment rédiger les tests unitaires

Lors de la rédaction des tests unitaires il faut vérifier les différents cas typiques d'utilisation de la classe, mais également les cas exceptionnels. Les tests aux limites seront présent ici. Il faut vérifier plusieurs scénarios de comportements.

Équivalence

Les tests d'*équivalence* permettent de vérifier que plusieurs méthodes ont un comportement équivalent. Cela permet de vérifier simultanément deux ou plusieurs méthodes. Pour les opérateurs de base, il existe plusieurs *patterns* standard à vérifier. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

```
(a+b)== (a+=b)
(a-b)== (a-=b)
(a*b)== (a*=b)
(a/b)== (a/=b)
(a<b)== (b>=a)
(a>b)== (b<=a)
    ...
```

Les écritures génériques (Page Écritures génériques) résolvent ces équivalences.

Inverse

Les tests d'*inverse* vérifient que les opérations contradictoires le sont bien. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

```
(a<b)!= (a>=b)
(a>b)!= (a<=b)
(a==b)!=(a!=b)
(!a) !=  (a)
    ...
```

Les écritures génériques (Page Écritures génériques) résolvent ces inverses.

Réversibilité

Les tests de *réversibilité* vérifient qu'une opération est réversible. Cela est très utile pour vérifier les commandes « Undo ». Les opérateurs arithmétiques sont également réversibles. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

```
a    ==(a+b-b)
a    ==(a-b+b)
a    ==(a*b/b)
a    ==(a/b*b)
a    ==(a+=b,a-=b)
a    ==(!(!a))
     ...
```

Ordres

Les tests d'*ordre* vérifient qu'une opération impacte correctement les relations d'ordre entre les objets. Pour toutes valeurs a, b et c, les tests suivants doivent être résolus :

```
si b>=0 alors a      <=       (a+b)
si b<=0 alors a      >=       (a-b)
(a<c)==(a<b && b<c)
     ...
```

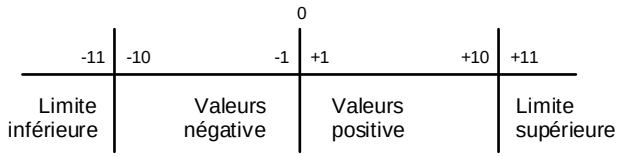
Associativité

Les tests d'*associativité* vérifient que l'ordre de rédaction d'une opération n'a pas d'impact sur le résultat. Pour toutes valeurs a et b, les tests suivants doivent être résolus :

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

Tests d'équivalence

Les *tests d'équivalence* vérifient des valeurs caractéristiques d'un ensemble de valeurs. Des tranches de valeurs peuvent être identifiées comme ayant le même comportement. Par exemple, tester une valeur négative peut être représentatif de toutes les valeurs négatives. Tester une valeur positive peut être également représentatif des valeurs positives. Les valeurs inférieures et supérieures aux limites peuvent également être testées par un représentant de ces valeurs.



Tester un représentant de chaque partition permet de tester l'ensemble des valeurs.

Tests aux limites

Les *tests aux limites* vérifient le comportement des méthodes avec des valeurs limites. Cela permet d'adapter les préconditions pour tenir compte des valeurs initiales valides entraînant un résultat final correct. La plupart des expressions de tests des chapitres précédents fonctionnent avec des valeurs normales pour les types de base du compilateur, mais échouent avec des valeurs limites. Par exemple, l'expression « `a<=(a+b)` » n'est pas vraie pour les entiers non signés, si `a` est différent de zéro et que `b` est égale à `UINT_MAX`. Une précondition de l'opérateur d'addition d'un entier devrait vérifier si le résultat attendu est compatible avec la valeur maximum d'un entier. Les tests unitaires vérifieront les expressions précédente avec des valeurs courantes, puis les mêmes tests seront effectués avec des valeurs aux limites. Par exemple, un test unitaire pour les objets `unsigned int` peut être rédigé comme cela :

```
enum CTstRet {Ok,Bad,TstCondition};

class CTstunsigned
{ unsigned* _a;
  unsigned* _b;
  public:
    CTstunsigned() : _a(NULL), _b(NULL) {}
  CTstRet ctra1()
  { if (_a!=NULL) return TstCondition;
    _a=new unsigned(10);
    return Ok;
  }
  CTstRet ctraLimitMax()
  { if (_a!=NULL) return TstCondition;
    _a=new unsigned(UINT_MAX);
    return Ok;
  }
  CTstRet dtra()
  { if (_a==NULL) return TstCondition;
    delete _a;
    _a=NULL;
    return Ok;
  }

  CTstRet ctrb1()
  { if (_b!=NULL) return TstCondition;
```

```
    _b=new unsigned(20);
    return Ok;
  }
  CTstRet ctrbLimitMax()
  { if (_b!=NULL) return TstCondition;
    _b=new unsigned(UINT_MAX);
    return Ok;
  }



  CTstRet dtrb()
  { if (_b==NULL) return TstCondition;
    delete _b;
    _b=NULL;
    return Ok;
  }

  CTstRet ordre()
  { if ((_a==NULL) || (_b==NULL)) return TstCondition;
    if ((*_b>=0) && !(*_a <= (*_a+*_b))) return Bad;
    if ((*_b>=0) && !(*_a >= (*_a-*_b))) return Bad;
    return Ok;
  }

  CTstRet scenario1();
  CTstRet scenario2();
};

CTstRet CTstunsigned::scenario1()
{ static CTstRet (CTstunsigned::* tab[])()=
  { &CTstunsigned::ctra1,&CTstunsigned::ctrb1,
    &CTstunsigned::ordre,
    &CTstunsigned::dtra,&CTstunsigned::dtrb
  };
  if ((_a!=NULL) || (_b!=NULL)) return TstCondition;
  CTstRet rc;
  for (int i=0;i<sizeof(tab)/sizeof(tab[0]);++i)
  { if ((rc=(this->*tab[i])())!=Ok) return rc;
  }
  return Ok;
}

CTstRet CTstunsigned::scenario2()
{ static CTstRet (CTstunsigned::* tab[])()=
  { &CTstunsigned::ctra1,&CTstunsigned::ctrbLimitMax,
    &CTstunsigned::ordre,
    &CTstunsigned::dtra,&CTstunsigned::dtrb
  };
  if ((_a!=NULL) || (_b!=NULL)) return TstCondition;
  CTstRet rc;
  for (int i=0;i<sizeof(tab)/sizeof(tab[0]);++i)
  { if ((rc=(this->*tab[i])())!=Ok) return rc;
  }
  return Ok;
}
```

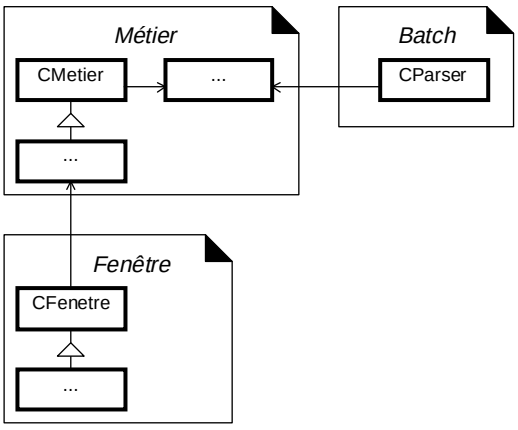
Le scénario « un » fonctionne correctement car les valeurs des objets sont normales. Le scénario « deux » ne fonctionne pas, car aux limites, les tests d'ordres ne sont plus valides. Les opérations sur les entiers non signés ne sont pas garanties !

g. Résumé

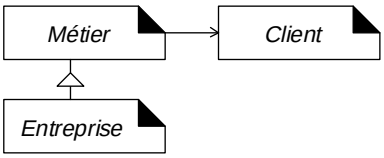
Les tests unitaires permettent de vérifier la rédaction des classes. Lors d'un développement avec une équipe importante, la plupart des classes doivent être simulées. Lors de l'intégration, les tests unitaires doivent être refaits après chaque ajout de nouveaux composants logiciels. Les erreurs éventuelles seront ainsi très rapidement localisées. La rédaction des tests unitaires permet au développeur de vérifier sa classe et également de savoir si les erreurs apparues en intégration viennent de celle-ci ou des autres composants logiciels précédemment simulés.

C. INTEGRATION

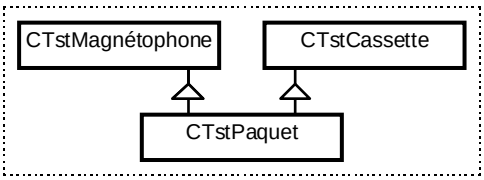
Après avoir passé avec succès les tests unitaires de chaque classe, il faut ensuite intégrer celles-ci dans des « paquets de classes ». Un paquet de classes est un ensemble de classes formant une couche logiciel. C'est une sorte de librairie. Cela peut correspondre à un `namespace`. Un paquet de classes est un regroupement de classes en relations fortes entres elles. Par exemple, les classes métier forment un paquet de classe. Ce paquet peut être utilisé par différents paquets qui offriront différentes interfaces aux classes du métier. Un paquet pourra par exemple offrir une interface fenêtre, et un autre offrira un langage auteur permettant de manipuler par un traitement par lot les objets du métier.



Un paquet peut lui-même utiliser d'autres paquets. Dans ce cas, il dépend de ceux-ci. Il existe des liens d'utilisations entre paquets. Un paquet peut également hériter d'un autre paquet si une de ces classes hérite d'une classe d'un autre paquet.

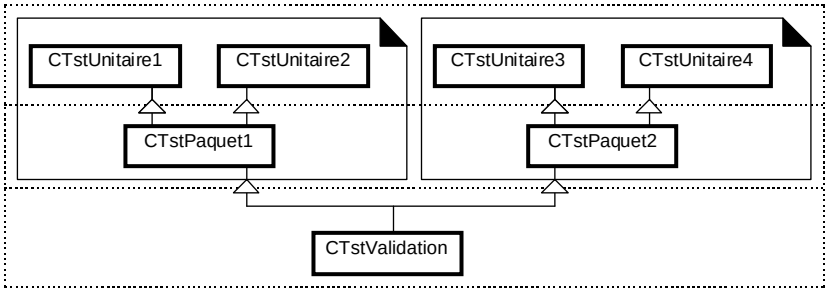


Lors des tests unitaires, les classes non encore intégrées sont simulées. Maintenant, il faut réunir les vraies classes et vérifier que les tests unitaires continuent à fonctionner. Il est possible qu'il faille modifier légèrement les tests pour pouvoir placer les classes précédemment simulées dans le bon état. Après avoir réuni les classes d'un paquet, il faut vérifier individuellement chaque classe, puis vérifier l'ensemble des classes du paquet. Pour cela, nous allons utiliser les tests unitaires de chaque classe. Nous allons par exemple déclarer la classe `CTstMagnetophone` pour tester la classe `CMagnetophone`, et la classe `CTstCassette` pour tester la classe `CCassette`. Nous allons écrire un test du paquet de classe héritant des deux classes de test unitaire.



Cela permet de bénéficier du contexte des deux tests. Il faut ensuite ajouter de nouveaux scénarios s'appuyant sur les tests de ces deux classes mais combinant les interactions entre celles-ci. L'héritage des classes de tests est optionnel. Cela est un confort, mais n'est pas obligatoire. Il est possible d'écrire un test d'un paquet de classes ayant son propre contexte qui n'a rien à voir avec les contextes des tests unitaires. Les scénarios des tests de paquet de classes utilisent la même technique que les tests unitaires. Les tests aléatoires sont possibles, soit en utilisant les méthodes de tests héritées, soit en utilisant les nouveaux tests rédigés au sein de la classe `CTstPaquet`.

Cette technique est applicable également lors de l'intégration de paquet de classes. Un test de l'intégration de plusieurs paquets de classes pourra hériter des tests de chacun des paquets. Un test de validation pourra hériter des tests de chacun des paquets de classes.



Une hiérarchie parallèle aux classes de l'application va progressivement être rédigée, permettant de tester sérieusement toute l'application.

Une fois l'exécutable terminé, il faut procéder aux tests a *posteriori*. Il faut dans un premier temps effectuer les tests fonctionnels, c'est-à-dire chercher à piéger le programme par une utilisation de celui-ci. Dans un deuxième temps, il faut effectuer un test de « stress ». Vérifiez que le programme supporte les charges prévues et a un comportement correct aux limites des ressources. Si la mémoire vient à manquer, le programme tolérera-t-il cela sans perte d'information ? Si le réseau est interrompu, les erreurs seront-elles correctement détectées et gérées ?

Il est également possible de rédiger des scénarios d'utilisations permettant de vérifier la non régression de l'application.

CHAPITRE 9

COMMENT ÇA MARCHE ?

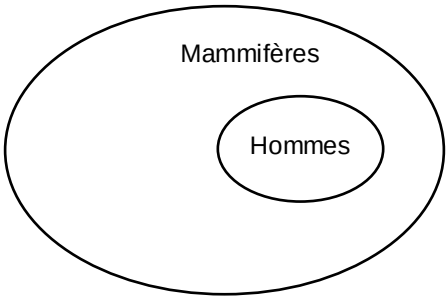
Pour bien comprendre les règles de développement indiquées dans les chapitres précédents, il me semble nécessaire de connaître les mécanismes utilisés par le compilateur pour générer le programme. Vous trouverez dans ce chapitre, la description de l'héritage et du polymorphisme qui sont les caractéristiques les plus connues du langage. D'autre part, vous trouverez une explication interne du mécanisme d'exception présent dans la nouvelle norme.

A. HERITAGES

Écrire un programme en suivant une approche objet consiste à spécifier des composants simples participant à une architecture plus complexe. L'ensemble de ces composants est en relation. L'héritage est une relation particulière (fort utile) proposée par les langages objets. Voici une description des différentes techniques utilisées pour générer les héritages.

a. Héritage simple

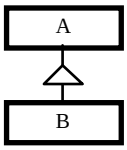
Dans le C++, l'héritage simple est une syntaxe permettant d'ajouter des comportements et des attributs à un composant simple. Dans une représentation ensembliste, cela correspond à un sous-ensemble ayant un comportement particulier. Par exemple, un `Homme` hérite des capacités de `Mammifère`.



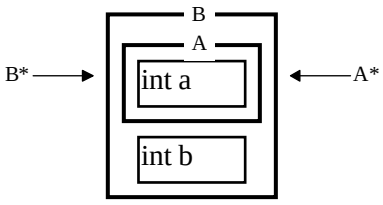
Toutes les propriétés de `Mammifère` sont héritées par `Homme`. Un `Homme` est une catégorie de `Mammifère`. Les attributs et les méthodes de `Mammifère` existent pour l'`Homme`.

Prenons un exemple d'héritage simple, et regardons comment le compilateur stocke l'objet en mémoire.

```
class A
{ public:
  int a;
};

class B : public A
{ public:
  int b;
};
```



L'objet est stocké en mémoire ainsi :

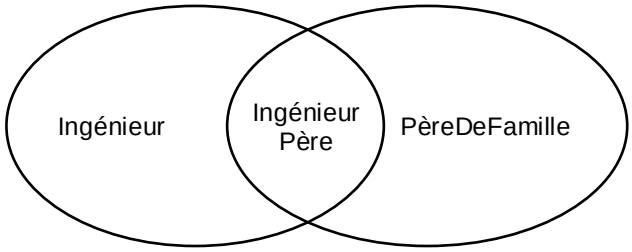


La structure B est accolée à la structure A. Un pointeur de type B est directement compatible avec un pointeur de type A. En langage C, cela se traduit comme cela :

```
struct A
{ int a;
};

struct B
{ struct A A;                              // Objet A hérité
  int b;
};
```

b. Héritage multiple

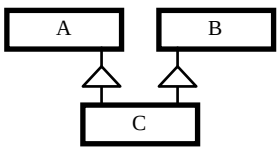
L'héritage multiple est une intersection entre plusieurs ensembles. Une classe hérite des capacités de plusieurs ensembles. Un homme peut être à la fois `Ingénieur` et `PèreDeFamille`.



Un `IngénieurPère` hérite des capacités de `Ingénieur` et des capacités de `PèreDeFamille`.

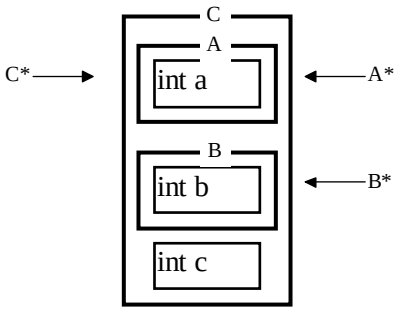
L'héritage multiple fonctionne comme précédemment, soit :

```
class A
{ public:
  int a;
};

class B
{ public:
  int b;
};


class C : public A,public B
```



```
{ public:
  int c;
};
```

L'objet est stocké en mémoire ainsi :



On remarque que l'objet C est construit en accolant les deux objets A et B. Le pointeur de l'objet C est compatible avec le pointeur de la partie A de C, mais pas avec le pointeur de la partie B de C. Cela montre qu'une conversion d'un pointeur C vers un pointeur B modifie la valeur du pointeur.

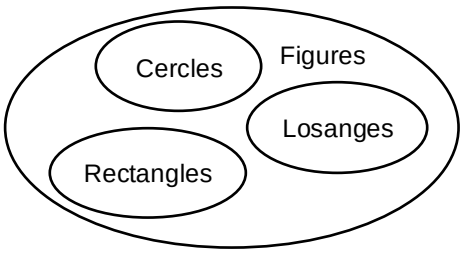
```
C* pc=&c;
B* pb=&c; // Ajuste le pointeur, conversion en pointeur B*
assert((void*)pb!=(void*)pc);
```

Cela explique pourquoi il faut faire attention lors des conversions. Les conversions par défaut ne posent pas de problème, car elles adaptent correctement la valeur du pointeur. Par contre, les conversions avec des pointeurs de type incompatible ne modifient pas la valeur du pointeur. Cela entraîne des erreurs très difficiles à détecter. En langage C, cela se traduit comme cela :

```
struct A
{ int a;
};

struct B
{ int b;
};

struct C
{ struct A A;                          // Objet A hérité
  struct B B;                          // Objet B hérité
  int c;
};
```

c. Méthodes virtuelles

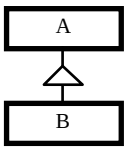
Le polymorphisme est la capacité de modifier certains comportements hérités. Une `Figure` géométrique peut être héritée par des `Cercles`, des `Rectangles`, des `Losanges`.



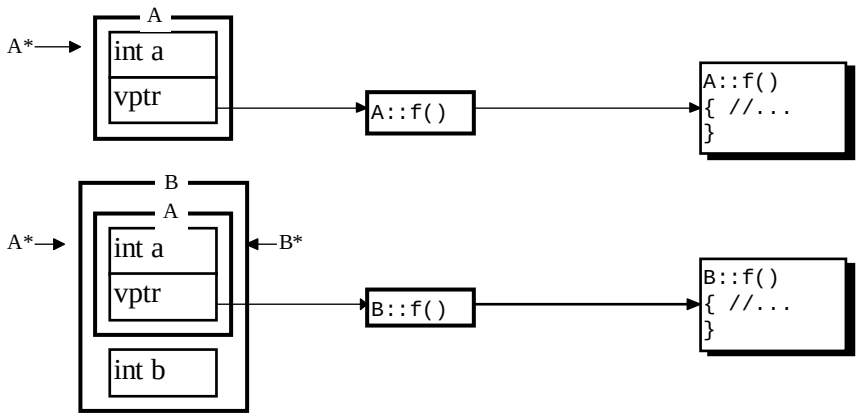
Une `Figure` offre la capacité de s'afficher. L'affichage d'un `Cercle` est différent de l'affichage d'un `Rectangle`. Chacun va vouloir modifier la méthode d'affichage de la `Figure` pour tenir compte des ses capacités propres d'affichage. Le polymorphisme permet à chaque classe de redéfinir un traitement hérité d'une classe de base.

Le polymorphisme est implanté à l'aide des méthodes virtuelles. Le compilateur construit une table de pointeur de fonctions pour chaque classe. Chaque instance de la classe possède un pointeur caché dans l'objet, pointant sur cette table. Ce pointeur est initialisé par le constructeur de l'objet.

```
class A
{ public :
  int a;
  virtual void f();
};

class B : public A
{ public:
  int b;
  virtual void f();
};
```



Les deux objets sont stockés en mémoire comme cela :



Le pointeur `vptr` pointe sur la table de saut. Lorsque la méthode `f()` est appelée pour un pointeur de type `A`, une indirection est effectuée pour appeler la méthode indiquée dans la table de saut. L'appel ressemble à cela :

```
A  a;
B  b;
A* pa=&a;
pa->vptr[0](); // Appel de A::f()
pa=&b;
pa->vptr[0](); // Appel de B::f()
```

On constate que le même appel peut aboutir à la version `A::f()` ou `B::f()`. Le polymorphisme est réglé par ce mécanisme. Chaque nouvelle méthode virtuelle est ajoutée à la suite de la table de saut. Un index lui est attribué. Le compilateur modifie chaque appel de méthode virtuelle par l'appel précédent, avec un index par méthode.

Le polymorphisme ne peut apparaître qu'avec des pointeurs ou des références. L'appel d'une méthode par l'opérateur « point », si l'objet n'est pas une référence, n'est pas ambigu. Dans ce cas, le compilateur peut appeler directement la méthode et utiliser éventuellement la version `inline` de celle-ci.

```
A a;
A* pa=&a;
a.f();   // Appel inline
pa->f(); // Appel non inline
```

Cela permet de générer en `inline` la méthode si nécessaire. Il n'est pas absurde de déclarer une méthode virtuelle en `inline`. Le compilateur choisit la technique d'accès la plus appropriée à l'utilisation.

Le pointeur `vptr` est initialisé par le constructeur. Mais attention, ce pointeur est initialisé lors de l'accolade ouvrante de ce dernier. Les constructeurs de `A` et de `B` sont traduits comme cela :

```
  A::A()
{ vptr=A::vtable;
}

  B::B()
{ A::A();
  vptr=B::vtable;
}
```

On constate que le pointeur `vptr` pointe sur la table `B::vtable` après l'appel du constructeur de `A`. Cela entraîne qu'un appel à une méthode virtuelle lors de l'exécution du constructeur de `A` appellera la méthode virtuelle de `A` mais pas la version virtuelle de `B`. Il ne faut pas appeler de méthode virtuelle dans les constructeurs (Règle CPP.DEB.14, page 184).

Les destructeurs initialisent le pointeur `vptr` avant de s'exécuter. Il n'est donc pas possible d'appeler une méthode virtuelle dans un destructeur. Les destructeurs de `A` et de `B` sont traduits comme cela :

```
  A::~A()
{ vptr=A::vtable;
  // Code du destructeur
}

  B::~B()
{ vptr=B::vtable;
  // Code du destructeur
  A::~A();
}
```

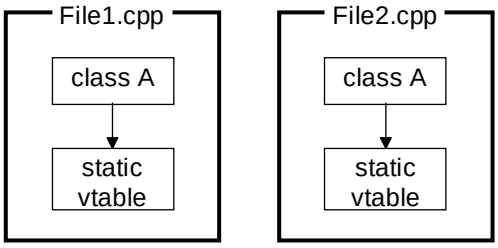
En langage C, cela se traduit ainsi :

```
typedef void (*fnvtable)();

fnvtable A::vtable[]=                    // Table de saut de A
{(fnvtable)A::f};

struct A
{ int a;
  fnvtable* vptr;                        // Pt de tbl de saut
};
// ctr
  A::A(A* const this)
```

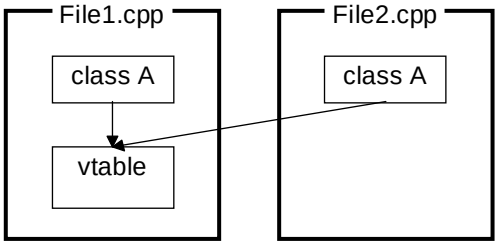
```
{ this->vptr=A::vtable;                    // Init vptr
}

// dtr
  A::~A(A* const this)
{ this->vptr=A::vtable;                    // Reset vptr
}

////////////////////////////////////////////////////////////////////
fnvtable B::vtable[]=                      // Table de saut de B
{(fnvtable)B::f};

struct B
{ struct A A;                              // Objet A hérité
  int b;
};

// ctr
  B::B(B* const this)
{ A::A(&this->A);
  this->A.vptr=B::vtable;                  // Modifie vptr de A
}

// dtr
  B::~B(B* const this)
{ this->A.vptr=B::vtable;                  // Reset vptr de A
  A::~A(&this->A);
}
```

Cette traduction en langage C n'est qu'un exemple de traduction possible. Les compilateurs sont libres de traduire le code C++ comme ils le veulent. En général, le pointeur `vptr` est ajouté à la fin de la structure C pour permettre une traduction aisée vers une structure C classique. L'ajout d'une méthode virtuelle à une structure ne modifie pas le début de la structure équivalente C. Une fonction C peut recevoir une structure C++ ayant une méthode virtuelle. Cela n'est pas obligatoire et dépend de chaque compilateur.

D'autre part, certains compilateurs créent la table de sauts virtuels en statique dans chaque module. Il y a autant de tables de sauts, qu'il y a de modules utilisant une méthode virtuelle de la classe.



D'autres génèrent un enregistrement particulier pour le `link` afin de partager les tables de sauts lors du `link`. Dans ce cas, il n'existe qu'une table de sauts par classe dans l'ensemble de l'application. Une autre approche consiste à générer la table de saut dans le fichier déclarant la première méthode virtuelle de la classe. Celle-ci ne devant être présente qu'une seule fois dans l'application, la table de saut l'est également.



Cela entraîne qu'il n'est pas possible de comparer les pointeurs de table de sauts virtuels entre deux instances d'une classe. Les pointeurs `vptr` ne sont pas forcément les mêmes, mais les fonctionnalités, elles, sont identiques. Il n'est pas possible de comparer deux instances ayant un `vptr` à l'aide de `memcmp()`.

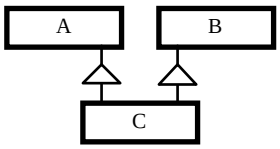
```
class A
{ int a;
  public:
  virtual void print();
  int operator ==(const A& x)
  { return !memcmp(this,&x,sizeof(x)); // Erreur
  }
};
```

d. Héritage multiple et méthodes virtuelles

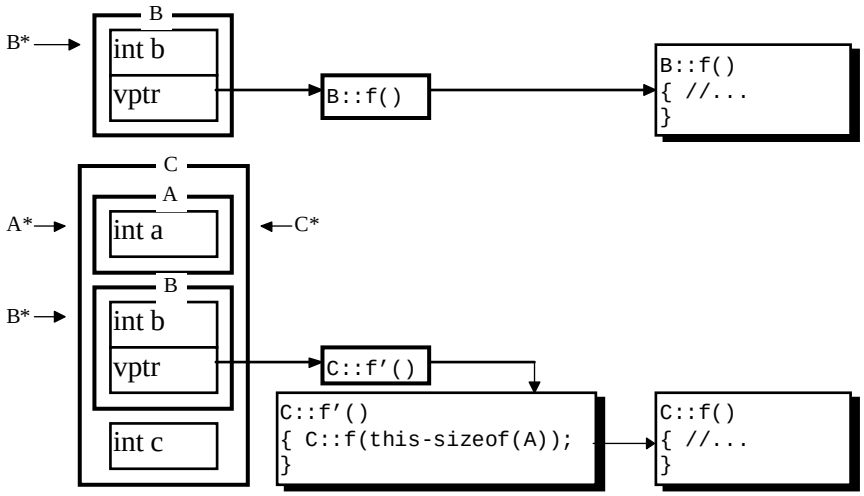
Une classe avec méthodes virtuelles

Si une classe hérite de plusieurs autres, dont une possède des méthodes virtuelles, il y a un problème d'adaptation de pointeur.

```
class A
{ public:
  int a;
};

class B
{ public:
  int b;
  virtual void f();
};

class C : public A,public B
{ public:
  int c;
  virtual void f();
};
```



La méthode `C::f()` doit recevoir un pointeur `this` sur un objet de type `C`, alors que la méthode `B::f()` attend un pointeur de type `B`. Comme nous l'avons vu plus haut, en cas d'héritage multiple, un pointeur de type `C` n'est pas directement compatible avec un pointeur de type `B`. Il faut ajuster le pointeur. Le pointeur étant ajusté sur la partie `B` de `C`, il faut l'ajuster de nouveau pour retourner à l'adresse de l'objet `C`. Le compilateur règle cela, en créant une méthode supplémentaire permettant d'ajuster la valeur de `this`. La mémoire est gérée comme décrit ci-après :



En langage C, cela se traduit ainsi :

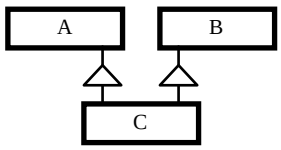
```
typedef void (*fnvtable)();

struct A
{ int a;
};

// ctr
  A::A(A* const this)
{
}

// dtr
  A::~A(A* const this)
{
}

//////////////////////////////////////////////////////////////////////
fnvtable B::vtable[]=                       // Table de saut de B
{(fnvtable)B::f};

struct B
{ int b;
  fnvtable* vptr;                           // Pt de tbl de saut
};

  B::B(B* const this)
{ this->vptr=B::vtable;                     // Init vptr
}

  B::~B(B* const this)
{ this->vptr=B::vtable;                     // Reset vptr
}

//////////////////////////////////////////////////////////////////////
struct C
{ struct A A;                               // Objet A hérité
  struct B B;                               // Objet B hérité
  int c;
};

fnvtable C::vtable[]=                       // Table de saut de C
{(fnvtable)C::f'};

void C::f'(B* const this)
{ C::f((C*)(((char*)this)-offsetof(C,B)));  // Convertie en (C*)this
}

// ctr
  C::C(C* const this)
{ A::A(&this->A);
  B::B(&this->B);
  this->B.vptr=C::vtable;                   // Modifie vptr de B
}
```

```
// dtr
  C::~C(C* const this)
{ this->B.vptr=C::vtable;                  // Modifie vptr de B
  B::B(&this->B);
  A::A(&this->A);
}
```

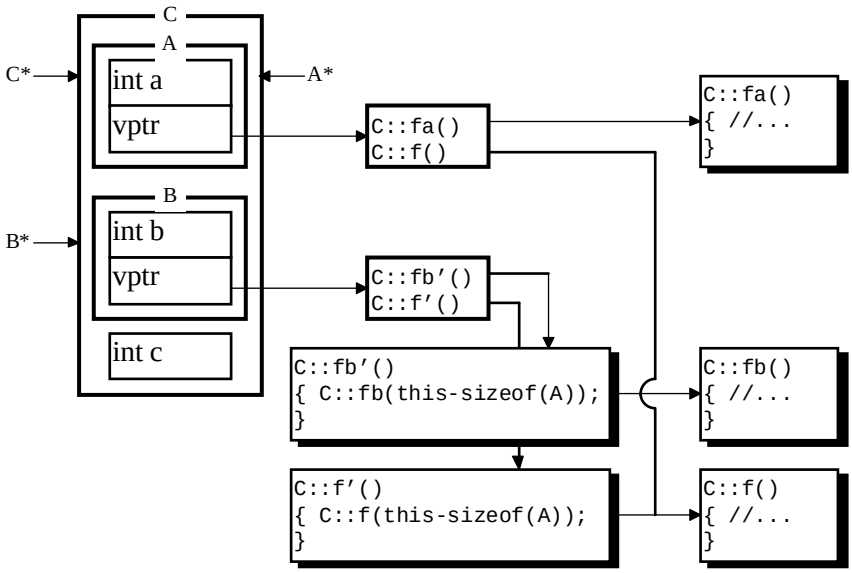
Plusieurs classes avec méthodes virtuelles

Lors d'un héritage multiple de classes ayant des méthodes virtuelles, il existe plusieurs pointeurs `vptr`.

```
class A
{ public:
  int a;
  virtual void fa();
  virtual void f();
};

class B
{ public:
  int b;
  virtual void fb();
  virtual void f();
};

class C : public A, public B
{ public:
  int c;
  virtual void fa();
  virtual void fb();
  virtual void f();
};
```



La mémoire est gérée comme cela :



Les méthodes supplémentaires ne sont ajoutées que pour les méthodes virtuelles des classes héritées en deuxième position et suivantes.

Si une méthode possède la même signature dans deux classes différentes, héritées par le même objet, la méthode virtuelle est indiquée dans les `vtable` des deux classes de base. C'est le cas de la méthode `f()`.

En langage C, cela se traduit comme cela :

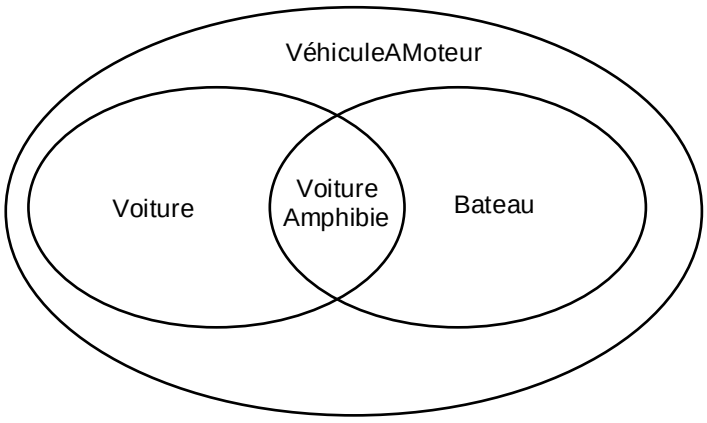
```
typedef void (*fnvtable)();

fnvtable A::vtable[]=                       // Table de saut de A
{(fnvtable)A::fa,
 (fnvtable)A::f
};

struct A
{ int a;
  fnvtable* vptr;                           // Pt de tbl de saut
};

// ctr
  A::A(A* const this)
{ this->vptr=A::vtable;                     // Init vptr
}
// dtr
```

```
  A::~A(A* const this)
{ this->vptr=A::vtable;                    // Reset vptr
}

////////////////////////////////////////////////////////////////////
fnvtable B::vtable[]=                      // Table de saut de B
{(fnvtable)B::fb,
 (fnvtable)B::f
};

struct B
{ int b;
  fnvtable* vptr;                          // Pt de tbl de saut
};

// ctr
  B::B(B* const this)
{ this->vptr=B::vtable;                    // Init vptr
}

// dtr
  B::~B(B* const this)
{ this->vptr=B::vtable;                    // Reset vptr
}

////////////////////////////////////////////////////////////////////
struct C
{ struct A A;                              // Objet A hérité
  struct B B;                              // Objet B hérité
  int c;
};

void C::fb'(B* const this)
{ C::fb((C*)(((char*)this)-offsetof(C,B)));// Convertie en (C*)this
}

void C::f'(B* const this)
{ C::f((C*)(((char*)this)-offsetof(C,B)));  // Convertie en (C*)this
}

fnvtable C::vtable1[]=                     // Table de saut de C pour A
{(fnvtable)C::fa,
 (fnvtable)C::f
};
fnvtable C::vtable2[]=                     // Table de saut de C pour B
{(fnvtable)C::fb',
 (fnvtable)C::f'
};

// ctr
  C::C(C* const this)
{ A::A(&this->A);
  this->A.vptr=C::vtable1;                 // Modifie vptr de A
  B::B(&this->B);
  this->B.vptr=C::vtable2;                 // Modifie vptr de B
}
```

```
// dtr
  C::~C(C* const this)
{ this->B.vptr=C::vtable2;                  // Modifie vptr de B
  this->A.vptr=C::vtable1;                  // Modifie vptr de A
  B::~B(&this->B);
  A::~A(&this->A);
}
```

e. Héritage virtuel

L'héritage virtuel est l'enrichissement le plus complexe. Un `Bateau` hérite de `VéhiculeAMoteur`. Une `Voiture` également. Une `VoitureAmphibie` hérite de `Voiture` et de `Bateau`. Si ce véhicule utilise le même moteur pour la propulsion lors de l'utilisation en `Voiture` et en `Bateau`, le moteur doit être partagé par les deux classes héritées. Dans une représentation ensembliste, cela donne :

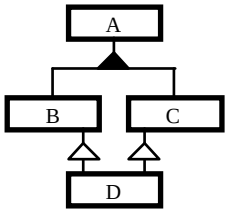


La voiture amphibie n'hérite qu'une seule fois des capacités de `VéhiculeAMoteur`.

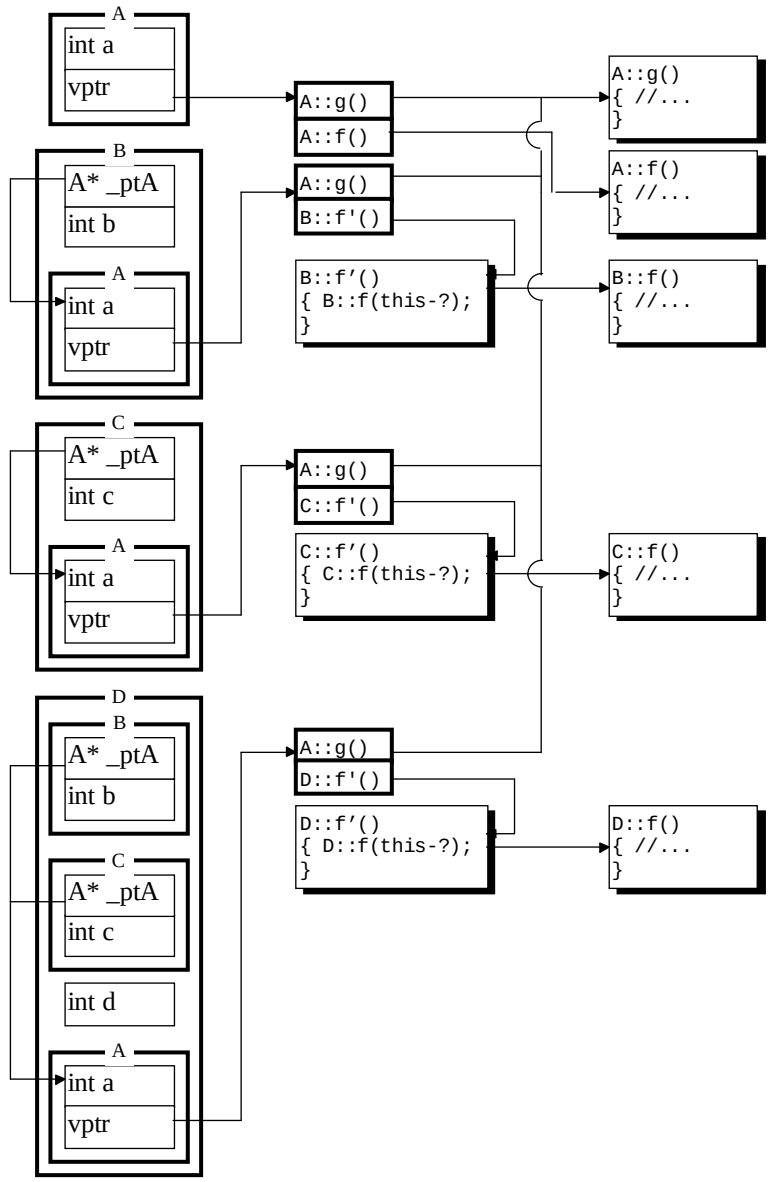
Vous trouverez une description complète de l'héritage virtuel dans le chapitre 10.1 du livre de A. Ellis et B. Stroustrup, « *The Annotated C++ Reference Manual* » (1994).

Voici un exemple simple.

```
class A
{ public:
  int a;
  virtual void g() { cout << "A::g()" << endl; }
  virtual void f() { cout << "A::f()" << endl; }
};

class B : virtual public A
{ public:
  int b;
  virtual void f() { cout << "B::f()" << endl; }
};

class C : virtual public A
{ public:
  int c;
  virtual void f() { cout << "C::f()" << endl; }
};

class D : public B, public C
{ public:
  int d;
  virtual void f() { cout << "D::f()" << endl; }
};
```



Le compilateur implante ces classes comme décrit ci-dessous :



La classe A possède un pointeur sur une table virtuelle. Cette table contient des pointeurs sur des méthodes. Pour un héritage simple, une autre table est simplement construite en ajustant les pointeurs des méthodes. Le problème se complique pour un héritage virtuel. En effet, dans l'exemple précédent, un pointeur de type B n'est pas directement compatible avec un pointeur de type A. Dans ce cas, il faut que le programme, lise le pointeur _ptA pour connaître l'adresse de l'objet A. Inversement, l'appel de la méthode f() à partir d'un pointeur de type A, doit ajuster le pointeur this pour qu'il pointe sur un objet du type de la nouvelle version de la méthode virtuelle. Dans l'exemple suivant,

```
void main()
{ D d;
  A* ptA=&d;
  ptA->f();

  D* ptD=&d;
  ptD->f();
}
```

l'appel de f() est effectué avec un pointeur this pointant sur la partie A de D, et non sur un objet de type D. Il faut ajuster ce pointeur pour le faire pointer sur un objet de type D. C'est le rôle des fonctions « prime ».

Celles-ci soustraient à this l'offset nécessaire puis appellent la version des méthodes correspondantes. Ces méthodes « prime » ne sont exécutées que lors d'un appel d'une méthode virtuelle d'une classe héritée.

En langage C, cela se traduit comme cela :

```
typedef void (*fnvtable)();

fnvtable A::vtable[]=                       // Table de saut de A
{(fnvtable)A::g,
 (fnvtable)A::f
};

struct A
{ int a;
  fnvtable* vptr;                           // Pt de tbl de saut
};

// ctr
  A::A(A* const this)
{ this->vptr=A::vtable;                     // Init vtable
}

// dtr
  A::~A(A* const this)
{ this->vptr=A::vtable;                     // Reset vtable
}
////////////////////////////////////////////////////////////////////////
```

```
struct _B                                   // Objet B sans A
{ A* _ptA;                                  // Pointeur sur A
  int b;
};

struct B
{ struct _B _B;                             // Objet B sans A
  struct A  A;                              // Objet A
};

void B::f'(A* const this)
{ B::f((B*)(((char*)this)-offsetof(B,A)));  // Convertie en (B*)this
}

fnvtable B::vtable[]=                       // Table de saut de B
{(fnvtable)A::g,
 (fnvtable)B::f'
};

// ctr
  _B::_B(_B* const this)
{ this->_ptA->vptr=B::vtable;               // Modifie vptr
}
  B::B(B* const this)
{ A::A(&this->A);                           // ctr de A
  this->_B._ptA=&this->A;                   // Init _ptA de _B
  _B::_B(&this->_B);                        // ctr de Obj B sans A
}

// dtr
  _B::~_B(_B* const this)
{ this->_ptA->vptr=B::vtable;               // Modifie vptr
}
  B::~B(B* const this)
{ this->_B._ptA=&this->A;                   // Reset _ptA de _B
  _B::~_B(&this->_B);                       // dtr de Obj B sans A
  A::~A(&this->A);                          // dtr de A
}

////////////////////////////////////////////////////////////////////////
struct _C                                   // Objet C sans A
{ A* _ptA;                                  // Pointeur sur A
  int c;
};
struct C
{ struct _C _C;                             // Objet C sans A
  struct A  A;                              // Objet A
};

void C::f'(A* const this)
{ C::f((C*)(((char*)this)-offsetof(C,A)));  // Convertie en (C*)this
}

fnvtable C::vtable[]=                       // Table de saut de C
{(fnvtable)A::g,
 (fnvtable)C::f'
```

```
};

// ctr
  _C::_C(_C* const this)
{ this->_ptA->vptr=C::vtable;              // Modifie vptr
}
  C::C(C* const this)
{ A::A(&this->A);                          // ctr de A
  this->_C._ptA=&this->A;                  // Init _ptA de _C
  _C::_C(&this->_C);                       // ctr de Obj C sans A
}

// dtr
  _C::~_C(_C* const this)
{ this->_ptA->vptr=C::vtable;              // Modifie vptr
}
  C::~C(C* const this)
{ this->_C._ptA=&this->A;                  // Reset _ptA de _C
  _C::~_C(&this->_C);                      // dtr de Obj C sans A
  A::~A(&this->A);                         // dtr de A
}

////////////////////////////////////////////////////////////////////
struct D
{ struct _B _B;                            // Obj B sans A
  struct _C _C;                            // Obj C sans A
  struct A  A;                             // Obj A
  int d;
};

void D::f'(A* const this)
{ D::f((D*)(((char*)this)-offsetof(D,A)));  // Convertie (D*)this
}

fnvtable D::vtable[]=                      // Table de saut de D
{(fnvtable)A::g,
 (fnvtable)D::f'
};

// ctr
  D::D(D* const this)
{ A::A(&this->A);                          // ctr de A
  this->_B._ptA=&this->A;                  // Init _ptA de _B
  _B::_B(&this->_B);                       // ctr Obj B sans A
  this->_C._ptA=&this->A;                  // Init _ptA de _C
  _C::_C(&this->_C);                       // ctr Obj C sans A
  this->A.vptr=D::vtable;                  // Modifie vptr
}

// dtr
  D::~D(D* const this)
{ this->A.vptr=D::vtable;                  // Reset vptr
  _C::~_C(&this->_C);                      // dtr Obj C sans A
  _B::~_B(&this->_B);                      // dtr Obj B sans A
  A::~A(&this->A);                         // dtr de A
}
```

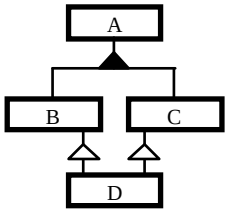
Ceci est la théorie. En réalité, pour éviter au maximum les méthodes « prime », les méthodes virtuelles sont générées avec un pointeur `this` pointant réellement sur un objet du type où la déclaration initiale de la méthode est effectuée. Dans le cas présent, `this` pointe sur la partie `A` de `D`, alors qu'il est vu par le programmeur comme étant de type `D`. Toute la compilation soustrait les offsets nécessaires pour accéder aux objets de type `D`. Cela veut dire qu'une même méthode `f()` rédigée normalement ou virtuellement est générée complètement différemment.

Malheureusement, cette technique n'est pas toujours possible. En effet, s'il existe une classe `E` héritant de la classe `D`, les éléments de `E` sont ajoutés à la fin de la structure `D`. Un pointeur sur `E` retrouve le même schéma que pour les méthodes « prime ». Dans ce cas, le compilateur ajoute les méthodes « prime ».

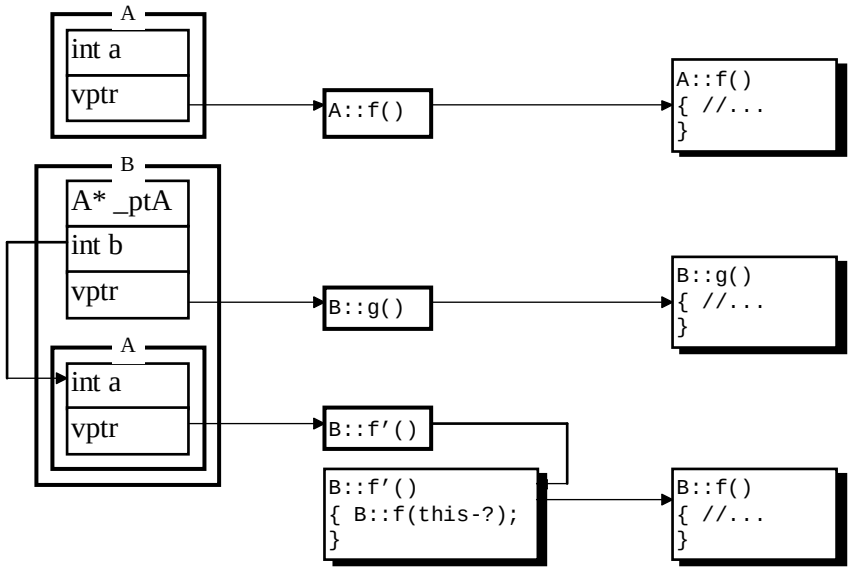
f. Héritages virtuels et méthodes virtuelles

Une classe dérivant virtuellement d'une classe de base et ajoutant de nouvelles méthodes virtuelles, ajoute un pointeur `vptr`.

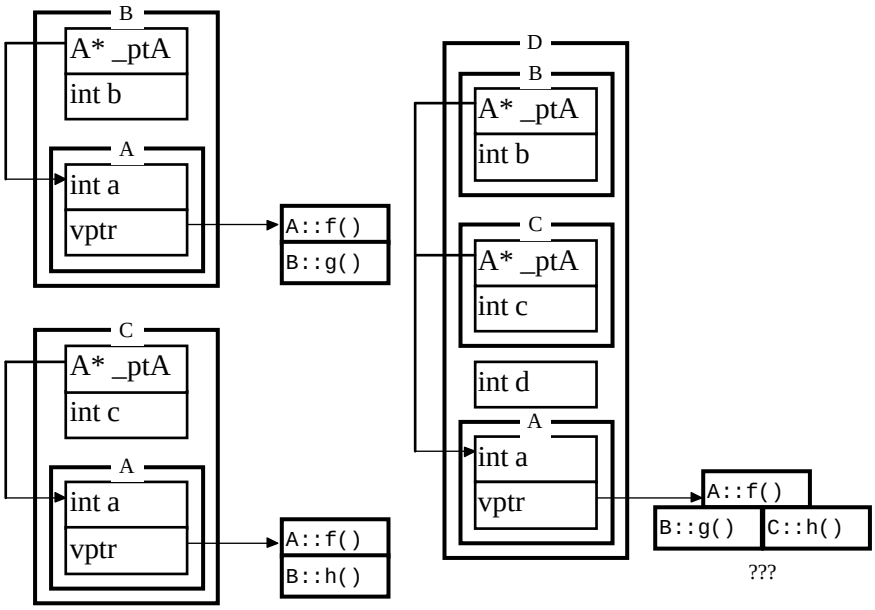
```
class A
{ public:
  int a;
  virtual void f() { cout << "A::f()" << endl; }
};

class B : virtual public A
{ public:
  int b;
  virtual void f() { cout << "B::f()" << endl; }
  virtual void g() { cout << "B::g()" << endl; }
};
```



Le compilateur implante ces classes comme cela :



L'appel de `g()` est effectué *via* le pointeur `vptr` de `B`. Le `vptr` de `A` ne peut pas être utilisé, car si une autre classe hérite virtuellement de `A` et déclare une autre méthode virtuelle `h()`, elle serait également en deuxième position dans la `vtable` de `A`. Deux méthodes différentes se retrouveraient à la même position.



Le compilateur ne pourrait pas construire la `vtable` de `D`. C'est pour cela qu'il faut obligatoirement ajouter un pointeur `vptr` pour toute nouvelle méthode virtuelle d'une classe dérivée virtuellement.

En langage C, cela se traduit comme cela :

```
typedef void (*fnvtable)();

fnvtable A::vtable[]=                    // Table de saut de A
{(fnvtable)A::f
};

struct A
{ int a;
  fnvtable* vptr;                        // Pt de tbl de saut
};

// ctr
  A::A(A* const this)
{ this->vptr=A::vtable;                  // Init vptr
}

// dtr
  A::~A(A* const this)
{ this->vptr=A::vtable;                  // Reset vptr
}
```

```
////////////////////////////////////////////////////////////////////
struct _B                                   // Objet B sans A
{ A* _ptA;                                  // Pointeur sur A
  int b;
  fnvtable* vptr;                           // Pt de tbl de saut
};

struct B
{ struct _B _B;                             // Objet B sans A
  struct A  A;                              // Objet A
};

void B::f'(A* const this)
{ B::f((B*)(((char*)this)-offsetof(B,A)));  // Convertie en (B*)this
}

fnvtable B::vtable1[]=                      // Table de saut de B pour A
{ (fnvtable)B::f'
};

fnvtable B::vtable2[]=                      // Table de saut de B
{ (fnvtable)B::g
};

// ctr
  _B::_B(_B* const this)
{ this->_ptA->vptr=B::vtable1;              // Modifie vptr de A
  this->vptr=B::vtable2;
}
  B::B(B* const this)
{ A::A(&this->A);                           // ctr de A
  this->_B._ptA=&this->A;                   // Init _ptA de _B
  _B::_B(&this->_B);                        // ctr de Obj B sans A
}

// dtr
  _B::~_B(_B* const this)
{ this->vptr=B::vtable2;
  this->_ptA->vptr=B::vtable1;              // Modifie vptr de A

}
  B::~B(B* const this)
{ _B::~_B(&this->_B);                       // dtr de Obj B sans A
  A::~A(&this->A);                          // dtr de A
}
```

g. Résumé

Pour conclure, voici un résumé des informations à connaître pour comprendre les principes de base de la traduction interne des héritages et du polymorphisme.

- L'héritage est traduit par l'accumulation des attributs des classes héritées.

- Le polymorphisme est géré à l'aide d'un pointeur caché, référençant une table de saut indiquant chaque méthode virtuelle. Ce pointeur sur la table de saut est initialisé dans le constructeur de la classe.
- Ce pointeur n'est jamais copié lors d'une affectation ou d'un constructeur de copie.
- La conversion d'un pointeur d'une classe dérivée vers un pointeur d'une classe de base peut modifier la valeur du pointeur.
- L'héritage virtuel utilise une indirection supplémentaire pour accéder à la classe de base.
- Il n'est pas possible de convertir un pointeur d'une classe de base héritée virtuellement en un pointeur d'une classe dérivée.

B. EXCEPTIONS

Les exceptions sont une extension importante introduite récemment dans le C++. Elles permettent de rédiger des programmes en ne s'occupant que du cas général, et de capturer les erreurs. Elles évitent de transporter un code d'erreur de fonctions en fonctions, pour remonter au traitement de celle-ci. De plus, il n'est pas possible de renvoyer un code d'erreur dans un constructeur. Les exceptions permettent de régler ce cas.

Tous les programmes doivent dorénavant capturer correctement les exceptions, et surtout, envisager la sortie impromptue d'une fonction lors de l'appel d'une autre. Lors de la génération d'une exception, la pile du programme est déroulée, et tous les destructeurs des objets présents dans la pile, lors de la remontée de celle-ci, sont appelés. Il faut rédiger les programmes, en envisageant la libération des ressources lors de l'appel de destructeur. Par exemple, le programme suivant :

```
void f()
{ char* pt=new char[10];

  strcpy(pt,"abc");
  g(pt);
  delete [] pt;
}
```

peut laisser de la mémoire perdue si la fonction `g()` génère une exception. Celle-ci n'étant pas capturée par la fonction `f()`, elle peut l'être par l'appelant de `f()`. Dans ce cas, le pointeur `pt` n'est jamais effacé. Il faut rédiger cette fonction en utilisant un pointeur d'agrégation (Voir « Durée de vie des objets », page 57).

```
void f()
{ CPtrArrayAggr<char> pt=new char[10];

  strcpy(pt,"abc");
  g(pt);
}
```

Avec cette nouvelle version, la ressource est libérée par le destructeur de `CPtrArrayAgr<char>`, donc en cas d'exception, elle est également libérée. Il faut dorénavant rédiger comme cela.

La future norme ANSI/ISO indique que l'opérateur `::new` ne retourne plus la valeur `NULL`, mais génère une exception. Ceci peut être désactivé, mais la tendance va dans ce sens.

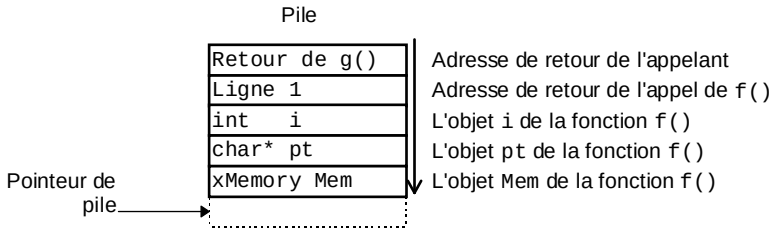
Comme on vient de le voir, l'appel d'une exception remonte la pile, en appelant l'ensemble des destructeurs des objets rencontrés.

Les exceptions s'utilisent comme cela :

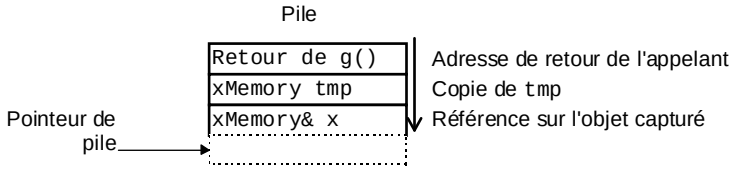
```
void f()
{ int i=10;
  char* pt=new char[i];
  if (pt==NULL)
  { xMemory Mem;
    throw Mem;
  }
}

void g()
{ try
  { f();                                    // 1
  } catch (xMemory& x)
  { // ...
  }
}
```

L'appel de `throw` ressemble à un appel de fonction. Le `catch` ressemble à la déclaration d'une fonction. La particularité des exceptions est que contrairement à un appel normal, le pointeur de pile du programme est réduit.

Avant l'appel du `throw`, la pile se présente comme cela :

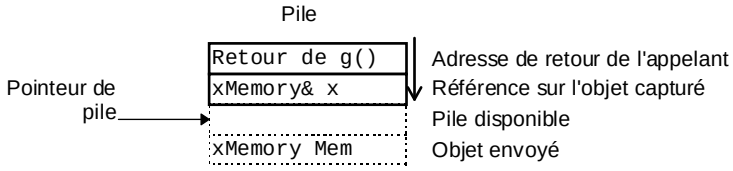


Le `throw` envoie un objet `xMemory` qui doit être capturé par le `catch`. Lors du `catch`, la pile doit être comme cela :



On constate que le pointeur de pile avant l'appel du `throw` ne permet pas de garder l'objet `Mem`. En effet, comme cet objet est stocké après l'adresse de retour de la fonction `f()`, celui-ci serait à une adresse dans la pile considérée comme vide lors de sa capture. C'est pour cela que l'objet envoyé dans une exception est toujours copié pour pouvoir être capturé.

Sans copie de l'objet, la pile serait comme cela :



La référence `x` pointerait sur l'objet `Mem` qui peut être détruit à tout moment par l'utilisation de la pile. Un appel de fonction ou une interruption matérielle détruirait l'objet `Mem`.

La copie de l'objet est effectuée avant l'appel des destructeurs des objets présents dans la pile lors de la remontée de celle-ci.

Si vous capturez un objet par valeur, celui-ci sera copié deux fois. Une première pour pouvoir être à une adresse valide lors de la capture, et une seconde fois pour obtenir cet objet par valeur lors du `catch`.

Pour éviter cette double copie de l'objet, il faut capturer un objet par référence.

Attention, celle-ci n'est pas la référence de l'objet envoyé, mais la référence de la copie. Cela entraîne que l'envoi d'un objet global sera capturé par une copie de celui-ci. La capture ne manipulera pas l'objet global mais la copie.

```
class CNetwork
{ public:
    CNetwork() { /*...*/ }
    CNetwork(const CNetwork& x) { /*...*/ }
    void Connect() { /*...*/ }
    void Deconnect() { /*...*/ }
};

CNetwork Network;

void f()
{ // ...
  throw Network;
}

void main()
{ try
  { f();
  }
  catch (CNetwork& x)
  { x.Deconnect(); }
}
```

Dans l'exemple précédent, la méthode `Deconnect()` n'est pas appelée pour l'instance globale `Network`, mais pour la copie de cette instance. Celle-ci peut demander une nouvelle connexion au réseau lors du constructeur de copie. C'est cette nouvelle connexion qui se fera déconnecter.

Pour recevoir l'objet lui-même, il faut envoyer son adresse et capturer celle de `CNetwork`.

```
void f()
{ // ...
  throw &Network;
}
void main()
{ try
  { f();
  }
  catch (CNetwork* x)
```

```
  { x->Deconnect(); }
}
```

Par contre, habituellement, recevoir une adresse dans un `catch` n'est pas recommandé.

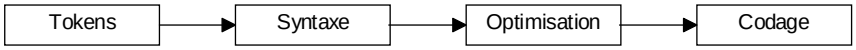
```
void f()
{ CNetwork Network;
  throw &Network; // Incorrecte
}
```

L'adresse de la variable automatique `Network` n'est plus valide lors du `catch`, l'objet `Network` étant détruit.

Une copie de l'objet envoyé, étant toujours effectuée, un objet n'ayant pas de constructeur de copie ne peut jamais être envoyé, et sera refusé par le compilateur.

C. INLINE

Comment le compilateur fait pour optimiser les appels d'une méthode `inline` ? Cela dépend, mais les principes généraux peuvent être décrits. Un compilateur analyse en plusieurs phases les sources. Dans un premier temps, une analyse lexicale permet d'identifier les différents éléments (*tokens*) du programme. Ensuite, une analyse syntaxique relie les éléments entre eux pour décrire les différentes étapes d'exécution. La troisième analyse simplifie l'arbre syntaxique. La dernière le traduit en code assembleur. En réalité, le compilateur exécute beaucoup plus d'étapes que celles décrites ici. Pour simplifier le discours, nous ne nous occuperons que de celles-ci.



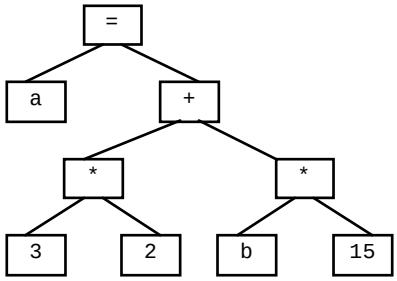
Pour éclairer ces étapes, nous allons prendre un exemple et décortiquer les premières phases du compilateur.

```
a=3*2+b*15;
```

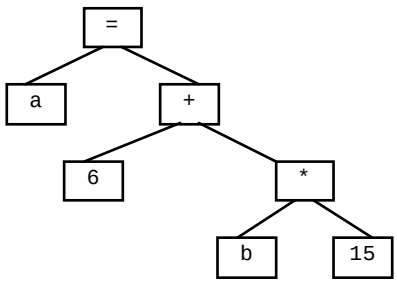
1. Identifications des *tokens*. Le compilateur détecte les *tokens* suivant :

'a' ; '=' ; '3' ; '*' ; '2' ; '+' ; 'b' ; '*' ; '15' ; ';'

2. Création de l'arbre syntaxique. En faisant abstraction du point virgule, l'arbre construit est le suivant :



3. Simplification de l'arbre. Le compilateur calcule toutes les constantes. L'arbre devient :



L'expression aurait pu être écrite comme ceci :

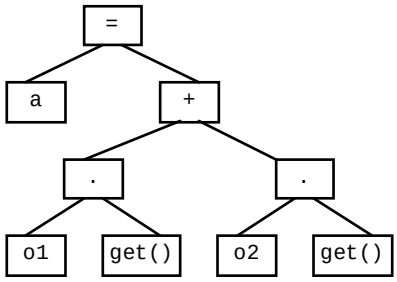
```
a=6+b*15;
```

le compilateur serait arrivé au même résultat. Ensuite, il traduit cet arbre par le langage machine spécifique à chaque unité centrale.

Comment cela se passe avec les méthodes `inline` ? Nous allons utiliser un autre exemple un peu plus compliqué que nous allons traduire en arbre.

```
class CObj
{ int _j;
  int _i;
  public:
    CObj(int i) : _i(i) {}
  int get() const;
};
int CObj::get() const { return _i; }

//...
{ CObj o1=4;
  CObj o2=7;
  int a;
  a=o1.get()+o2.get();                    // expression
}
```

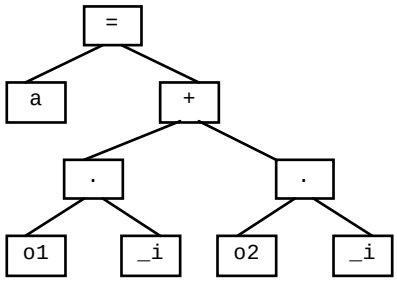
L'expression est traduite par un arbre équivalent à celui-ci :



Pour chaque appel de méthode le compilateur génère :

- la sauvegarde dans la pile de l'adresse `this`,
- l'ajout des paramètres de la méthode dans la pile,
- l'appel de la méthode
- la suppression des paramètres et de `this` de la pile.

Même si la méthode ne comporte aucun traitement, le compilateur va générer toutes les étapes indiquées ci-dessus. Pour notre exemple, le compilateur va générer deux appels à la méthode `get()`. Si cette méthode est déclarée en `inline`, l'arbre d'appel est différent.

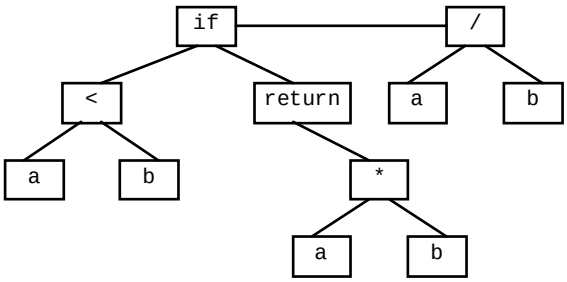


Il n'y a plus d'appel de méthode. Le compilateur utilise directement les adresses de `o1` et `o2` pour consulter leurs paramètres `_i`. L'appelant aura autant de copies de celui-ci que d'appels. Le programme sera plus long, mais le nombre d'instructions exécutées sera inférieur. Le coût de l'appel d'une fonction, quelque soit son corps est non nul. Si le corps de l'instruction a un coût inférieur à la procédure d'appel, il est préférable de déclarer la méthode en `inline`. Dans notre exemple, déclarer la méthode `get()` avec l'attribut `inline` permet d'accélérer le programme et de réduire sa taille ! Le coût d'appel d'une méthode ne peut pas être connu à priori. Ce coût est différent suivant les microprocesseurs et les compilateurs. En général, une méthode « courte » de deux ou trois lignes peut être déclarée en `inline`.

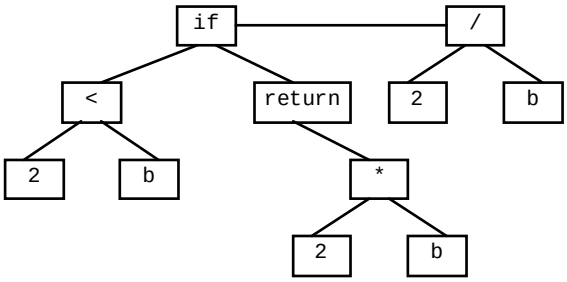
D'autre part, nous avons vu plus haut que le compilateur simplifie l'arbre syntaxique en précalculant les constantes. Un appel d'une méthode avec une constante en paramètre peut également augmenter les chances d'optimisation si la méthode est `inline`. En effet, le compilateur va générer chaque appel par une nouvelle version de la méthode, mais en remplaçant le paramètre constant par sa valeur. L'arbre syntaxique aura ainsi plus de chance d'être optimisé. Toute méthode est traduite en un arbre syntaxique. Cet arbre est recopié à chaque appel en y remplaçant les paramètres. Ensuite le compilateur optimise cet arbre.

```
inline int calcul(int x,int y)
{ if (x<y) return x*y;
  return x/y;
}
//...
void f(int a,int b)
{ calcul(a,b);           // Expr 1
  calcul(2,b);           // Expr 2
  calcul(3,4);           // Expr 3
  calcul(4,2);           // Expr 4
  calcul(a,calcul(3,4)); // Expr 5
}
```

Pour les cinq appels de la fonction `calcul`, les arbres d'appels deviennent :



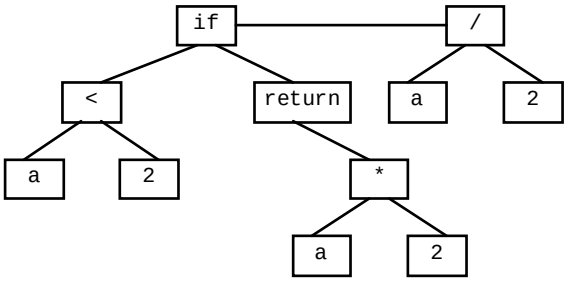
Expression 1 : `calcul(a,b);`



Expression 2 : `calcul(2,b);`

Expression 3 : `calcul(3,4);`

Expression 4 : `calcul(4,2);`



Expression 5 : `calcul(a,calcul(3,4));`

On constate que l'arbre peut être complètement différent suivant les paramètres reçus en entrée. Si une méthode ou une fonction reçoit une constante, et que celle-ci peut avoir un grand effet simplificateur sur l'arbre syntaxique, il est judicieux de déclarer la méthode en `inline`.

Certains compilateurs refusent de traduire une fonction en `inline` si celle-ci possède des boucles ou est trop importante pour pouvoir être gardée en mémoire par le compilateur. Dans ce cas, il fait abstraction de cet attribut, et les appels sont générés de façon classique. D'autres compilateurs, au contraire, n'imposent aucune limite à la déclaration d'une méthode `inline`. Dans ce cas de figure, une méthode très importante, appelée rarement et avec une constante particulière pour chaque appel, peut être un bon candidat à la génération `inline`. Le temps de compilation sera très important et le programme imposant en mémoire, mais l'exécution sera très efficace. Il n'est pas rare que le compilateur traduise une succession d'appels imbriqués de fonctions `inline`, par la simple lecture d'un attribut. Vous serez certainement surpris par les possibilités d'optimisations offertes par ces méthodes. Les compilateurs peuvent énormément réduire la taille de l'arbre syntaxique. Il ne faut pas se priver de rédiger des méthodes toutes petites pour accéder à un attribut car elles seront toutes diluées dans l'arbre syntaxique final.

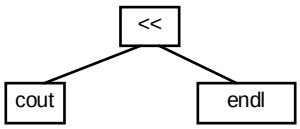
Par exemple, les flux du C++ utilisent des « manipulateurs ». Ceux-ci sont implantés à l'aide de pointeurs de fonction. Lors d'un appel de :

```
cout << endl;
```

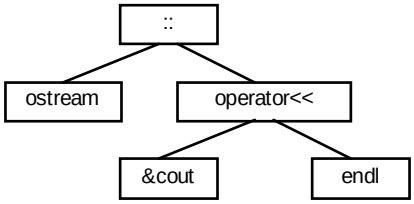
le compilateur utilise l'`operator <<()` recevant un pointeur de fonction. Ce pointeur est valorisé par l'adresse de `endl`. Cet opérateur appelle simplement la fonction reçue en paramètre.

```
inline ostream& ostream::operator <<(ostream& (*manip)(ostream&))
{ return ((*manip)(*this)); }
```

L'arbre syntaxique de cet appel est le suivant :



Il est ensuite modifié comme ceci :



ce qui correspond à une écriture équivalente à :

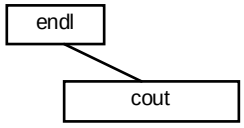
```
cout.operator <<(endl);
```

ou bien, en indiquant explicitement le paramètre `this` :

```
ostream::operator <<(&cout,endl); // Ce n'est pas du C++
```

`&cout` est le pointeur `this` de l'`operator <<()` et `endl` est le paramètre.

Comme le code de l'opérateur est `inline`, l'arbre devient après simplification :



L'appel de `endl` est effectué directement. L'`operator <<()` à disparu de l'arbre. Si l'expression avait été :

```
endl(cout);
```

L'arbre syntaxique aurait été le même. Ce n'est pas fini ! Comme le manipulateur `endl` est lui aussi `inline`, le code de celui-ci est généré *in situ* lors de la compilation de

l'expression. Contrairement aux fonctions `printf` du C, l'utilisation des flux du C++ est *compilé* grâce à l'existence des méthodes `inline`. Le code le plus efficace est généré pour chaque utilisation. Si vous appelez l'opérateur ci-dessus avec un code comme ci-après :

```
void f(ostream& (*manip)(ostream&))
{ cout << manip;                          // expr
}
void main()
{ f(endl);
}
```

la compilation de l'expression est différente. En effet, le compilateur crée une version non `inline` de la fonction `endl()`, puis fournit l'adresse de celle-ci à `f()`. L'expression appelle directement la fonction `endl` par l'intermédiaire du paramètre `manip`. Le code `endl` n'est pas généré en `inline` dans l'expression contrairement à l'exemple précédent. Par contre, si la fonction `f()` est `inline`, l'appel de `endl` est dilué entièrement dans l'appelant.

Lorsque vous rédigez votre programme, essayez de prévoir l'arbre syntaxique compris par le compilateur. Certains experts rédigent leurs programmes en fonction des possibilités d'optimisations. Par exemple, un code comme celui-ci :

```
enum Direction {Nord,Ouest,Sud,Est};
Direction const Directions[]={Nord,Ouest,Sud,Est};

// ---------- Autre fichier ----------
inline void recherche(Direction x)
{ //...
}

void main()
{ for (int i=0;i<4;++i)
  { recherche(Directions[i]);
  }
}
```

utilise un tableau de constantes pour manipuler l'ensemble des quatre points cardinaux. La boucle `i` appelle la fonction `recherche` pour chacune des constantes. Une seule génération de cette fonction est effectuée dans le corps de la boucle. Cette traduction utilise une variable pour le paramètre de `recherche`. Le compilateur ne peut pas précalculer les expressions de la fonction `recherche` lors de sa génération `inline`. Si le programme est rédigé différemment :

```
enum Direction {Nord,Ouest,Sud,Est};
Direction const Directions[]={Nord,Ouest,Sud,Est};
```

```
// ---------- Autre fichier ----------
inline void recherche(Direction x)
{ //...
}

void main()
{ recherche(Nord);
  recherche(Ouest);
  recherche(Sud);
  recherche(Est);
}
```

le compilateur va pouvoir spécialiser la fonction `recherche` pour chacune des constantes d'orientation. Le choix de la rédaction de `main()` est ici dicté par les possibilités d'optimisation du compilateur. Le corps de la fonction `main()`sera beaucoup plus gros, car quatre générations de `recherche()` seront effectuées. Par contre, l'exécutable sera beaucoup plus rapide. Vous pouvez également modifier les signatures de vos méthodes pour augmenter les chances d'utilisation de constantes. Par exemple, un programme de jeux voulant parcourir un damier suivant les directions horizontales et verticales peut utiliser une méthode recevant dans les paramètres le coefficient à ajouter à la coordonnée `X`, et le coefficient pour la coordonnée `Y` au lieu de recevoir la direction voulue.

```
const int MaxX=10;
const int MaxY=10;
struct
{ int offsetx;
  int offsety;
} Offset[]={{1,0},{0,1}};

enum TDirection {Horizontal, Vertical};

void calcul(TDirection dir)
{ int offx=Offset[dir].offsetx;
  int offy=Offset[dir].offsety;
  for (int x=0;x<MaxX;x+=offx)
  { for (int y=0;y<MaxY;y+=offy)
    { //...
    }
  }
}
void main()
{ calcul(Horizontal);
  calcul(Vertical);
}
```

peut être modifié comme cela :

```
const int MaxX=10;
const int MaxY=10;
inline void calcul(int offx,int offy)
```

```
{ for (int x=0;x<MaxX;x+=offx)
  { for (int y=0;y<MaxY;y+=offy)
    { //...
    }
  }
}
void main()
{ calcul(1,0);
  calcul(0,1);
}
```

Ce c ode est moins élégant que le précédent, mais le compilateur pourra l'optimiser au mieux. La signature de `calcul()` est modifiée pour pouvoir bénéficier d'une meilleure optimisation.

Pour corriger votre programme, il ne faut surtout pas que les méthodes appelées soient diluées dans l'arbre d'appel. Vous ne retrouveriez plus vos petits... C'est pour cela que tous les compilateurs C++ offrent une option permettant de rejeter tous les attributs `inline`. Le compilateur se comportera comme si ces attributs n'avaient jamais existé. Cela vous permet de suivre pas à pas votre programme. Ensuite lors de la phase d'intégration, en changeant les paramètres de compilation, vous pourrez générer une version nettement plus rapide et souvent moins exigeante en taille mémoire !

Voici décrits quelques principes vous permettant d'imaginer le comportement du compilateur lors de la phase d'optimisation :

- la manipulation d'une variable globale est plus rapide que la manipulation d'une variable locale ou d'un paramètre,
- la manipulation d'une constante est plus rapide que la manipulation d'une variable,
- le compilateur effectue tous les calculs sur les constantes,
- un appel de fonction ou de méthode est moins rapide que l'exécution directe de celle-ci dans l'appelant,
- un appel de méthode virtuelle est moins rapide qu'un appel de fonction classique.

D. CONCLUSION

Le C++ est un langage extrêmement riche mais difficile à maîtriser correctement. Souvent à cause de sa richesse d'utilisation, il y a beaucoup de pièges camouflés. Tout au long de ce livre, vous avez pu en apprécier les finesses et les manques. On peut en effet regretter qu'il n'y ait pas de syntaxe particulière pour :

- séparer la sémantique des quatre types de pointeurs (Page 57),
- les attributs virtuels (Page 51),
- les relations virtuelles (Page 53),
- gérer correctement les invariants et les pré et postconditions (Page 237),
- ...

Ce langage est jeune, beaucoup d'évolutions arrivent. Peut-être qu'une version future comblera les lacunes actuelles et réduira les risques d'erreurs. Une démarche dans ce sens a déjà été entreprise en offrant différents mécanismes de conversion pour y ajouter une sémantique vérifiable par le compilateur. Nous entrons dans l'ère du C++ comme nous avons eu l'ère du C. Gageons que le C++ ne sera pas détrôné avant longtemps. Malgré ses faiblesses, j'espère que vous prendrez plaisir à maîtriser ce langage dans toutes ses finesses. Le C++ est un langage non trivial. Les développeurs prennent généralement plus de satisfaction à se mesurer à des problèmes difficiles qu'à accumuler les résolutions de problèmes triviaux. Le C++ devrait apporter de ce point de vue une réelle satisfaction.

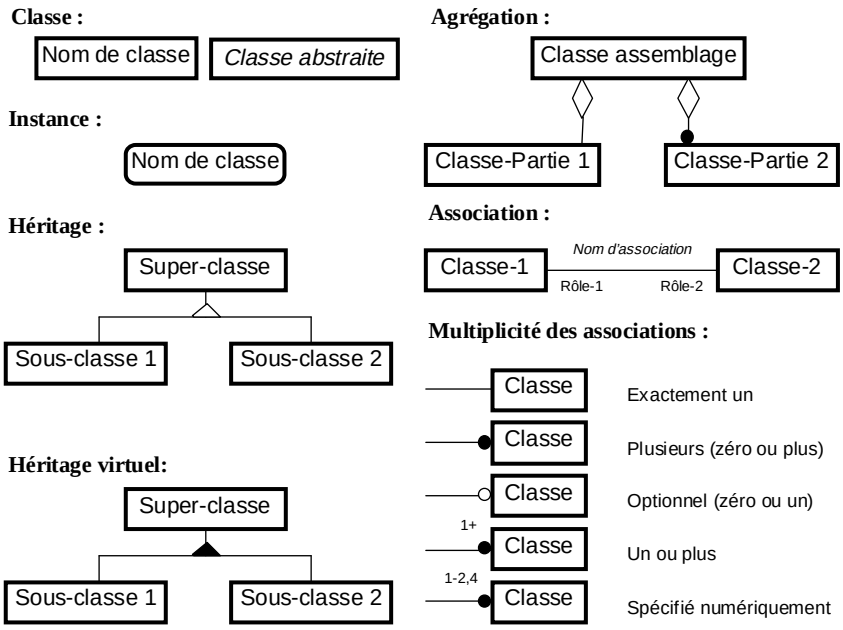
ANNEXES

A. NOTATION OMT

Voici un extrait de la notation objet OMT (*Object Modeling Technique*) telle qu'elle est utilisée dans cet ouvrage. Cette notation, proposé par Rumbaugh, est beaucoup plus riche que l'extrait indiqué ici. OMT est une méthode de modélisation par objet. Une démarche uniforme est utilisée pour l'ensemble de l'analyse d'une application. Trois modèles sont utilisés conjointement pour décrire et analyser correctement une application objet ou non :

- le modèle objet,
- le modèle dynamique,
- le modèle fonctionnel.

L'extrait suivant ne concerne que le modèle objet.



B. DIFFERENCES ENTRE LE C ET LE C++

Le C++ est une évolution du C ANSI. La compatibilité est très forte entre ces deux langages, mais elle n'est pas parfaite. Il existe quelques différences minimes qu'il est utile de connaître afin de faciliter le portage de sources C ANSI [CR].

- Il existe un nouveau type de commentaires.

```
int n=4//* commentaire */
       -2;
```

  En C, `n=4/-2` ; en C++ `n=4-2`.

- De nouveaux mots clefs sont ajoutés.

```
int public=4; /* valide C, invalide C++ */
```

- Le type des caractères littéraux est modifié d'`int` en `char`.

```
sizeof('A')==sizeof(int) // Valide C, invalide C++
```

- Les structures ont une visibilité locale.

```
struct A { int x; }
void f()
{
  struct B
  { struct A
    { float x;
    } a;
  };
  struct A a; /* A interne en C, A extern en C++ */
  a.x=0;      /* float en C, int en C++ */
}
```

- La vue d'un fichier pour une constante est en link interne si elle n'est pas déclarée `extern`.

```
const int n=0; /* Link interne en C++, extern en C */
```

- `main` ne peut pas être appelé récursivement et on ne peut obtenir son adresse (Conséquence de l'initialisation des objets `static`).
- C accepte des types compatibles que le C++ n'accepte pas.

```
void f();
void f(int); /* deux fonctions en C++, même fonction en C */
```

- La conversion de `void*` vers un pointeur sur objet demande une conversion.

```
void* pv=0;
char* pc=pv; /* Valide en C, invalide en C++ */
```

- Un pointeur sur un objet `const` ne peut pas être implicitement converti en `void*`.

```
const int n=0;
void* pv=&n; /* Valide en C, invalide en C++ */
```

- La déclaration implicite est interdite.

```
int main() { return f(); } /* Valide en C, invalide en C++ */
```

- Les types doivent être déclarés en dehors des expressions.

```
p=(void*)(struct x {int i; } *)0; /* Valide en C, invalide en C++ */
```

- Il est interdit de sauter au-delà d'une déclaration initialisée.

```
int fn(int x,int y)
{ if (x<y) goto Label;
  Mode b=Fixe;
Label:
  return y;
}
```

  Cela se traduit dans les `switch`.

```
enum Mode {Fixe,Mobile} x;
/*...*/
switch(x)
{ case 1 :
    Mode b=Fixe; /* Valide en C, invalide en C++ */
    /*...*/
    break;
/*...*/
}
```

- Une fonction non `void`, doit renvoyer une valeur.

```
int fn(int* x)
{ if (x) ++*x;
} /* Valide en C, invalide en C++ */
```

- Les attributs `static` et `extern` ne peuvent être employés que pour le nom d'objets ou de fonctions.

```
extern struct Node {...}; /* Valide en C, invalide en C++ */
```

- Les objets `const` doivent être initialisés.

```
const int obj; /* Valide en C, invalide en C++ */
```

  Par contre, ANSI C définit la tentative de définitions.

```
int obj;    /* Tentative definition */
int obj=10; /* Definition en C, invalide en C++ */
```

- Un objet de type énuméré ne peut prendre que les valeurs de l'énumération.

```
enum couleur {rouge,vert,bleu};
couleur c=1; /* Valide en C, invalide en C++ */
```

- Le type d'une constante d'énumération est du type de l'énumération et non du type entier.
- Une fonction déclarée sans paramètre est vue du type `(void)`.

```
int fn() /* Equivalent à int fn(void) en C++, et int fn(...) en C */
```

- Les types ne peuvent pas être déclarés lors d'un retour de fonction.

```
struct CRet
{ /*...*/
} /* Attention, il manque le ';' */

/* Valide en C, Invalide en C++ */
  f()
{ /*...*/ }
```

- La syntaxe du C++ refuse l'ancienne syntaxe du C K&R.

```
int swap(x,y)
int *x,*y; /* Valide en C, invalide en C++ */
{ /*...*/}
```

- La déclaration d'un tableau de caractères avec une chaîne pour sauter le caractère '`\0`' est impossible.

```
char array[4]="abcd"; /* Valide en C, invalide en C++ */
```

- Le nom des classes est local à la classe de déclaration.

```
struct X
{ struct Y
  { /*...*/
  } y;
};
struct Y yy; /* Valide en C, invalide en C++ */
```

- Le nom d'un `typedef` ne peut pas être redéfini dans la déclaration d'une classe.

```
typedef int I;
struct S
{ I i;
  int I; /* Valide en C, invalide en C++ */
};
```

- L'adresse d'une variable `register` est illégale.

```
register int i;
int* ip=&i; // Invalide en C, valide en C++
```

- `NULL` est à « `(void*)0` » en C et `0` en C++.

```
void f(const char* format,...);
f("P",NULL);        // Valide en C, invalide en C++
f("P",(void*)NULL); // Valide en C et C++
```

C. TABLE DE PRIORITES

Niveau	Opérateur
1 Droite	`::` unaire
1 Gauche	`::` binaire
2 Gauche	`-> .`
2 Gauche	`[]`
2 Gauche	`()` appel de fonction
2 Gauche	`()` constructeur de type
3 Droite	`sizeof`
3 Droite	`++ --`
3 Droite	`~`
3 Droite	`!`
3 Droite	`+ -` unaire
3 Droite	`*` & adressage
3 Droite	`()` cast
3 Droite	`new delete`
4 Gauche	`->* .*`
5 Gauche	`* / %`
6 Gauche	`+ -` binaire
7 Gauche	`<< >>`
8 Gauche	`< <= >= >`
9 Gauche	`== !=`
10 Gauche	& et binaire
11 Gauche	~ xor binaire
12 Gauche	| ou binaire
13 Gauche	&& et logique
14 Gauche	|| ou logique
15 Gauche	`?:`
16 Droite	`= *= /= %= += -= <<=`
17 Droite	`>>= &= |= ^=`
18 Gauche	`,`

D. LEXIQUE

Adoption Récupération par un objet d'un pointeur pour en devenir le propriétaire. L'objet devient alors responsable de la destruction de l'objet qu'il adopte.

Affectation Copie d'un objet dans un autre. Les objets sont déjà créés. Cela s'effectue à l'aide de l'`operator =()`.

Agrégation Relation particulière entre deux objets. L'un *possède* l'autre. Si l'effacement du propriétaire entraîne la destruction de tous les objets en relation, c'est une agrégation. Les attributs d'une classe sont des agrégations. La destruction de la classe entraîne la destruction de ses attributs.

Attribut Un attribut est un élément caractérisant un objet. Ce peut être un objet dans un objet (une instance d'une classe, un entier, ou tout type de base). L'agrégation et les relations se traduisent par des attributs.

Base (classe de) Une classe de base est une classe n'héritant pas d'une autre classe mais étant elle-même héritée.

Cast Voir « Conversion ».

Constructeur Opération de construction ou d'initialisation d'un objet.

Container Voir « Contenant ».

Conteneur Objet ayant pour fonction de contenir d'autres objets. Un tableau est un « Conteneur », une liste chaînée également.

Conversion Action de transformation d'un objet d'un certain type vers un autre type.

Dérivée (classe) Une classe dérivée est une classe héritant d'une autre classe.

Fonction Une « fonction » est un traitement ne dépendant pas d'un objet, contrairement aux méthodes.

Héritage Déclaration permettant de récupérer les fonctionnalités d'un objet pour les modifier ou les compléter.

Instance Une « instance » est un exemplaire d'une classe d'objets. Chaque création d'objet entraîne la création d'une

	« instance » de la classe.
Méthode	Une « méthode » est un traitement appartenant à une classe. Une méthode possède un pointeur `this` contrairement aux fonctions.
Polymorphisme	Le « polymorphisme » est la capacité d'adaptation d'un comportement générique à différents objets d'une hiérarchie de classes. Une méthode de même nom sera interprétée différemment suivant la classe de l'objet.
Portée	La « portée » est la visibilité des variables dans un contexte donné. La « portée » globale correspond aux variables globales. La « portée » d'une fonction correspond aux paramètres et aux variables déclarées dans la fonction. La « portée » d'une méthode correspond aux attributs d'un objet ainsi qu'aux attributs statiques de la classe.
Prototype	Le « prototype » est une déclaration de la syntaxe d'une méthode ou d'une fonction sans rédiger le corps de celle-ci.
Sémantique	La « Sémantique » est le sens de quelque chose. Certains opérateurs ont un sens courant qu'il convient de respecter.
Scope	Voir « Portée ».
Surcharge	La « surcharge » est la possibilité d'utiliser un même nom de fonction ou de méthode avec des paramètres différents. Les paramètres permettent au compilateur de connaître la version à appeler.
Signature	La « signature » d'une fonction ou d'une méthode est le prototype complet de celle-ci. Deux méthodes ayant le même nom mais des paramètres différents ont des signatures différentes.

E. WORLD WIDE WEB

Pour obtenir des informations supplémentaires sur le C++, voici quelques adresses Internet utiles.

- Draft du standard ANSI/ISO C++

  ftp ://research.att.com/dist/c++std/WP/

- The standard Template Library

  http ://www.cs.rpi.edu/~musser/stl.html

- Librairies libres ou commerciales C++

  http://www.quadralay.com/www/CCForum/CCLibrary.html

F. REFERENCES

1. [Stroustrup, ARM:94] *The Annotated C++ Reference Manual*, 1994, de Bjarne Stroustrup et A. Ellis, Addison Wesley (ISBN: 0-201-51459-1). Le document de base du comité de normalisation ANSI/ISO du langage.

2. [Stroustrup, DE:94] *The Design and Evolution of C++*, 1994, de Bjarne Stroustrup, Addison Wesley (ISBN: 0-201-54330-3). L'historique de la genèse du langage et les évolutions rejetées.

3. [Plauger, DStLib:95] *The draft standard C++ library*, 1995, de P.J. Plauger, Prentice Hall PTR (ISBN: 0-13-117003-1). Une implantation de la future norme des librairies C++.

4. [Rumbaugh et al, OMT:95] *OMT Modélisation et conception orientées objet*, 1995, de James Rumbaugh et al., Masson (ISBN: 2-225-84684-7). Traduction française de la norme OMT.

5. [Desfray, OE] *Object Engineering*, de Philippe Desfray, Addison-Wesley (ISBN: 0-201-42288-3). Norme de conception « Classe-Relation ».

6. [Meyer,CPO:90] *Conception et programmation par objet : pour du logiciel de qualité*, 1990, de Bertrand Meyer, Interéditions (ISBN: 2-72-9602-720).

7. [Gamma et al, DP:94] *Design Patterns*, 1994, de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley (ISBN: 0-201-63361-2). Modèles de conceptions pour tous usages.

8. [Graham et al, SI:93] *Software Inspection*, 1993, de Dorothy Graham et Tom Gilb, Addison-Wesley (ISBN: 0-201-631-814). Techniques de tests.

9. [Stepanov et al, STL:95] *The Standard Template Library*, 1995, de Alexander Stepanov et Meng Lee. Documentations des conteneurs standardisés.

10. [CR] *C++ Report*, SIGS Publication. Magazine Américain traitant exclusivement du C++.

11. [JOOP] *Journal of Object-oriented programming*, SIGS Publication. Magazine Américain traitant des langages objets.

INDEX