De C++ à Objective-C

version 2.1 fr

Pierre Chatelier
e-mail : pierre.chatelier@club-internet.fr

Adaptation en langue anglaise : Aaron Vegh

Révisions de ce document disponibles à l'adresse :
`http://pierre.chachatelier.fr/programmation/objective-c.php`

Ce document est aussi disponible en anglais
This document is also available in English

Remerciements: Pour leurs lectures attentives et leurs multiples remarques, je tiens avant tout à remercier Pascal Bleuyard, Jérôme Cornet, François Delobel et Jean-Daniel Dupas, en l'absence de qui l'écriture de ce document aurait été parfaitement possible, mais en aurait largement pâti. Jack Nutting, Ben Rimmington et Mattias Arrelid ont également soumis de nombreuses observations par la suite. Jonathon Mah, en particulier, a apporté un remarquable lot de corrections très pertinentes.
Ces personnes ne sont pas responsables des erreurs que j'ai pu rajouter après leur relecture.

Table des matières

Introduction

Ce document est un guide de passage de C++ à Objective-C. Il existe plusieurs documentations soucieuses d'enseigner le modèle objet *via* Objective-C, mais aucune à ma connaissance n'est destinée aux codeurs expérimentés en C++, désirant se renseigner sur les concepts du langage pour les comparer à ce qu'ils connaissent déjà. Le langage Objective-C m'avait semblé au premier abord un obstacle plutôt qu'un tremplin à la programmation avec **Cocoa** (cf. section 1.1 page suivante) : il est si peu répandu que je ne comprenais pas son intérêt face à un C++ puissant, efficace et maîtrisé. Il a donc fallu longtemps pour que je comprenne qu'il était au contraire un réel concurrent grâce à la richesse des concepts qu'il propose. Ce document ne se présente pas comme un didacticiel mais comme une référence de ces concepts. Il permettra ainsi, je l'espère, d'éviter qu'une mauvaise connaissance d'Objective-C conduise un développeur C++, soit à abandonner trop vite ce langage, soit à utiliser à mauvais escient ses outils habituels, produisant alors un code bâtard, inélégant et inefficace. Ce document ne se veut pas une référence *complète*, mais *rapide*. Pour une description approfondie d'un concept, mieux vaut consulter une documentation Objective-C spécifique [4].

La comparaison avec le C# nécessiterait un autre document, car ce langage est beaucoup plus proche de l'Objective-C que le C++. Un programmeur C# s'adapte sans doute assez facilement à l'Objective-C. D'après moi, le C#, malgré des concepts avancés, est tout de même très inférieur à l'Objective-C, à cause de sa complexité pour des choses simples d'accès en Objective-C, et une API `.NET` très loin de la qualité de Cocoa. Cet avis est personnel et son argumentation dépasse le cadre du présent document.

1 Objective-C

1.1 Objective-C et Cocoa

Une première précision me semble indispensable : **Objective-C** est un langage, et **Cocoa** est un ensemble de classes permettant de programmer de façon native sous MacOS X. En théorie, on peut faire de l'Objective-C sans Cocoa : il existe un *front-end* pour GCC. Sous MacOS X, les deux sont presque indissociables, la plupart des classes fournies avec le langage faisant en réalité partie de Cocoa.

Pour être précis, Cocoa est l'implémentation par Apple, pour MacOS X, du standard OpenStep publié par NeXT Computer en 1994. Il s'agit d'une bibliothèque de développement d'applications basée sur Objective-C. Citons l'existence de GNUstep [6], une autre implémentation, libre, d'OpenStep. Cette implémentation se veut la plus portable possible pour fonctionner avec de nombreux systèmes. Elle est toujours en développement au moment où j'écris ces lignes.

1.2 Bref historique d'Objective-C

Il est assez difficile de dater précisément les naissances des langages de programmation. Selon que l'on considère leurs balbutiements, leur développement, leur annonce officielle ou leur standardisation, la marge est importante. Malgré tout, un historique sommaire, présenté en figure 1, permet de situer Objective-C vis-à-vis de ses ancêtres et ses concurrents.

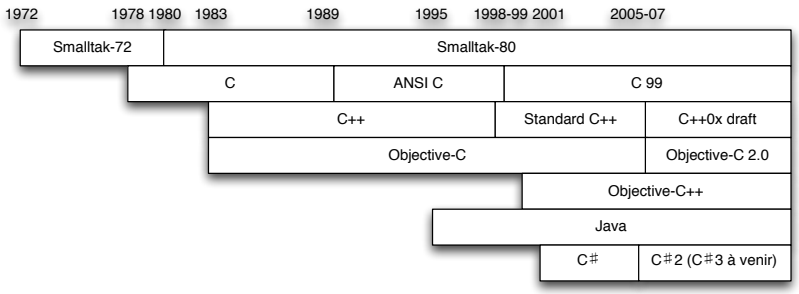

FIGURE 1 – Historique sommaire de Smalltalk, Java, C, C#, C++ et Objective-C

Smalltalk-80 est un des tous premiers langages réellement objet. Le C++ et l'Objective-C sont deux branches différentes visant à créer un langage objet basé sur le C. Objective-C emprunte énormément à Smalltalk, pour le dynamisme et la syntaxe, tandis que C++ se tourne vers une forme de programmation très statique orientée vers la performance d'exécution. Java veut se placer en remplaçant du C++ mais il est également très inspiré de Smalltalk par son aspect objet plus pur. C'est pourquoi, malgré le titre de ce document, de nombreuses références sont également faites au Java. Le langage C#, développé par Microsoft, est un concurrent direct d'Objective-C.

Objective-C++ est une fusion de l'Objective-C et du C++. Il est déjà en grande partie opérationnel, mais certains comportements sont encore imparfaits à la date d'écriture de ces lignes. Objective-C++ doit permettre de mélanger des syntaxes Objective-C et C++ pour tirer parti des fonctionnalités des deux mondes (cf. section 14 page 67).

1.3 Objective-C 2.0

Le présent document a été mis à jour pour tenir compte des modifications apportées par Objective-C 2.0, introduit avec MacOS 10.5. Bien qu'importants techniquement, les nouveaux outils qu'apportent ces modifications aux développeurs s'énumèrent assez rapidement. Il s'agit :

- du ramasse-miettes : cf. section 6.6 page 50 ;
- des propriétés (*properties*) : cf. section 12.2 page 61 ;
- de l'énumération rapide : cf. section 11.2.2 page 58 ;
- des mots-clefs `@optional` et `@required` pour les protocoles : cf. section 4.4 page 23 ;
- de la mise à jour des fonctions de manipulation bas niveau de la bibliothèque *run-time* d'Objective-C : cf. section 13.2 page 66.

Chacune de ces nouveautés est détaillée dans une section spécifique.

2 Généralités sur la syntaxe

2.1 Mots-clefs

Objective-C est un sur-ensemble du langage C. Comme en C++, un programme correctement écrit en C devrait être compilable en Objective-C, sauf s'il utilise certaines mauvaises pratiques autorisées par le C. L'Objective-C n'a fait que rajouter des concepts et les mots-clefs les accompagnant. Pour éviter tout conflit, ces mots-clefs commencent par le caractère @ (at). En voici la (courte) liste exhaustive : `@class`, `@interface`, `@implementation`, `@public`, `@private`, `@protected`, `@try`, `@catch`, `@throw`, `@finally`, `@end`, `@protocol`, `@selector`, `@synchronized`, `@encode`, `@defs` (qui n'est plus documenté dans [4]). Objective-C 2.0 (cf. 1.3 page précédente) a ajouté `@optional`, `@required, @property`, `@dynamic`, `@synthesize`. Notons également la présence des valeurs `nil` et `Nil`, du type `id`, du type `SEL` et du type `BOOL` avec ses valeurs `YES` et `NO`. Enfin, il existe quelques mots-clefs qui ne sont définis que dans un certain contexte bien particulier, et qui ne sont pas réservés hors de ce contexte : `in`, `out`, `inout`, `bycopy`, `byref`, `oneway` (ceux-là peuvent se rencontrer dans la définition de protocoles : cf. section 4.4.5 page 25) et `getter`, `setter`, `readwrite`, `readonly`, `assign`, `retain`, `copy`, `nonatomic` (ceux-là peuvent se rencontrer dans la définition des propriétés : cf. section 12.2 page 61).

Il y a une très forte confusion possible entre les mots-clefs du langage et certaines méthodes héritées de la classe racine `NSObject` (la mère de toutes les classes, cf. section 3.1 page 10). Par exemple, les apparents « mots-clefs » de la gestion mémoire que sont `alloc`, `retain`, `release` et `autorelease`, sont en fait des méthodes de `NSObject`. Les mots `super` et `self` (cf. section 3.3.1 page 13), pourraient également passer pour des mots-clefs, mais `self` est en fait un argument caché de toute méthode, et `super` une directive demandant au compilateur d'utiliser `self` différemment. Il ne me paraît cependant pas préjudiciable de faire l'amalgame entre ces faux « mots-clefs » et les vrais, vu leur indispensable présence.

2.2 Commentaires

Les commentaires `/* ... */` et `//` sont autorisés.

2.3 Mélange code/déclarations

Il est possible comme en C++ d'insérer des déclarations de variables au milieu d'un bloc d'instructions.

2.4 Nouveaux types et valeurs

2.4.1 BOOL, YES, NO

En C++, le type `bool` a été implémenté. En Objective-C, on dispose du type `BOOL` qui prend les valeurs `YES` ou `NO`.

2.4.2 `nil`, `Nil` et `id`

Ces trois mots-clefs sont expliqués dans la suite de ce document. On peut toutefois les présenter brièvement :

- Tout objet est de type `id`. C'est un outil de typage faible ;
- `nil` est l'équivalent de `NULL` pour un pointeur d'objet. `nil` et `NULL` ne sont pas interchangeables ;
- `Nil` est l'équivalent de `nil` pour un pointeur de classe, car en Objective-C, les classes sont aussi des objets (instances de méta-classes).

2.4.3 `SEL`

`SEL` est le type utilisé pour stocker les *sélecteurs*, obtenus par l'utilisation de `@selector`. Un sélecteur représente une méthode sans objet associé : on peut l'utiliser comme équivalent à un pointeur de méthode. Ce n'est cependant pas, techniquement, un pointeur de fonction. Voyez la section 3.3.5 page 16 pour des explications à ce sujet.

2.4.4 `@encode`

Dans un souci d'interopérabilité, les types de données en Objective-C, même les types personnalisés et les prototypes de fonctions et méthodes, peuvent être encodés en caractères ASCII, selon un format documenté [4]. Un appel à `@encode(`*`un type`*`)` envoie une chaîne au format C standard (`char*`) représentant le type.

2.5 Organisation du code source : fichiers *.h*, fichiers *.m* et inclusion

Comme en C++, il est bienvenu d'utiliser un couple de fichiers interface/implémentation par classe. En Objective-C, le fichier d'en-tête est un fichier *.h* et le code est dans un fichier *.m* ; on rencontre aussi l'extension *.mm* pour l'Objective-C++, qui fait l'objet d'une section spéciale en fin de document (section 14 page 67). Enfin, notons que l'Objective-C introduit la directive de compilation `#import` pour remplacer avantageusement `#include`. En effet, tout fichier d'en-tête en C doit posséder des gardes de compilation pour empêcher son inclusion multiple ; le `#import` gère cela automatiquement. Ci-après se trouve un exemple typique de couple interface/implémentation. La syntaxe Objective-C est expliquée dans la suite du document.

C++

```
//Dans le fichier Toto.h

#ifndef __TOTO_H__ //garde de
#define __TOTO_H__ //compilation

class Toto
{
...
};

#endif
```

```
//Dans le fichier Toto.cpp

#include "Toto.h"

...
```

Objective-C

```
//Dans le fichier .h

//déclaration de classe
//(différent du «interface» de Java)
@interface Toto : NSObject
{
...
}
@end
```

```
//Dans le fichier Toto.m

#import "Toto.h"

@implementation Toto
...
@end
```

2.6 Nom des classes : pourquoi NS ?

Dans ce document, la plupart des classes sont précédées de *NS*, comme *NSObject*, *NSString*... La raison en est simple : ce sont des classes Cocoa, et la plupart des classes Cocoa commencent par *NS* puisqu'elles ont été initiées sous NeXTStep.

C'est une pratique courante que d'utiliser un tel préfixe afin d'identifier l'origine des classes.

2.7 Différenciation entre fonctions et méthodes

Objective-C n'est pas un langage « dont les appels de fonction s'écrivent avec des crochets ». C'est en effet ce que l'on pourrait penser en voyant écrit

```
[object doSomething];
```

au lieu de

```
object.doSomething();
```

D'une part, comme Objective-C est un sur-ensemble du C, les *fonctions* respectent la même syntaxe et la même logique que le C quant à la déclaration, l'implémentation et l'appel. En revanche, les *méthodes*, qui n'existent pas en C, ont une syntaxe spéciale avec des crochets. De plus, la différence ne se situe pas seulement au niveau de la syntaxe mais également de la sémantique. Cela est expliqué plus en détails dans la section 3.2 page suivante : ce n'est pas un appel de méthode, c'est l'envoi d'un message. Même si l'effet est semblable pour la structure du code, l'implémentation de ce mécanisme permet plus de souplesse et de dynamisme. Il est par exemple compatible avec l'ajout de méthodes pendant l'exécution (cf. section 13.2 page 66). La syntaxe est aussi beaucoup plus claire, surtout en cas d'appels en cascade (cf. section 3.3.1 page 13).

3 Classes et objets

Objective-C est un langage objet : on y crée des classes et des objets. Il respecte cependant mieux le paradigme objet que C++, lequel présente des lacunes vis à vis du modèle idéal. Par exemple, en Objective-C, les classes sont elles-mêmes des objets manipulables dynamiquement : on peut, pendant l'exécution du programme, ajouter des classes, créer des instances d'une classe d'un certain nom, demander à une classe quelles méthodes elle implémente, etc. Tout cela est beaucoup plus puissant que les RTTI du C++ (cf. section 13.1 page 65), qui ont été greffées à un langage fondamentalement « statique ». On déconseille d'ailleurs souvent de les utiliser, car les résultats obtenus dépendent du compilateur lui-même, et ne garantissent aucune portabilité.

3.1 Classe racine, type `id`, valeurs `nil` et `Nil`

Dans un langage objet, on crée généralement un diagramme de classes pour chaque programme. Une des particularités d'Objective-C par rapport à C++ est l'existence d'une classe racine (*root*) dont devrait hériter toute nouvelle classe. En Cocoa, il s'agit de `NSObject`, qui fournit un nombre de services gigantesque, et assure la cohérence du système d'exécution d'Objective-C. Mais la notion de classe racine n'est pas une spécifité d'Objective-C, c'est une particularité du modèle objet en général ; Smalltalk, Java, utilisent aussi une classe racine. Le C++ quant à lui ne l'impose pas.

En toute rigueur, tout objet devrait donc être de type `NSObject` et tout pointeur vers un objet pourrait ainsi être déclaré comme un `NSObject*`. Cependant, ce n'est pas absolument obligatoire : il existe le type `id`, court et pratique pour exprimer un pointeur vers un objet quelconque, et qui implique une vérification dynamique des types, et non plus statique. Cela peut être utile comme outil de typage faible sur des méthodes génériques. Notez aussi qu'un pointeur d'objet nul ne devrait pas être mis à `NULL` mais à `nil`. Ces valeurs n'ont pas vocation d'être interchangeables. Un pointeur C normal peut être à `NULL`, mais `nil` a été introduit en Objective-C pour les pointeurs d'objets. En Objective-C, les classes sont aussi des objets (instances de méta-classes), et il existe donc des pointeurs de classes, pour lesquels la valeur vide est `Nil`.

3.2 Déclaration de classes

Il est difficile de montrer par un seul exemple toutes les différences existant entre le C++ et l'Objective-C pour la déclaration des classes et l'implémentation des méthodes. En effet, syntaxe et concepts s'entremêlent et nécessitent des explications. Les différences sont donc exposées en plusieurs étapes très ciblées.

3.2.1 Attributs et méthodes

En Objective-C, les attributs sont appelés *données d'instance*, et les fonctions membres des *méthodes*.

C++	Objective-C

`class Toto`
`{`
`  double x;`
 	
`  public:`
`  int  f(int x);`
`  float g(int x, int y);`
`};`
 	
`int  Toto::f(int x) {...}`
`float Toto::g(int x, int y) {...}`
	`@interface Toto : NSObject`
	`{`
	`  double x;`
	`}`
	
	`-(int)  f:(int)x;`
	`-(float) g:(int)x :(int)y;`
	`@end`
	
	`@implementation Toto`
	`-(int)  f:(int)x {...}`
	`-(float) g:(int)x :(int)y {...}`
	`@end`

En C++, attributs et méthodes sont déclarés ensembles au sein de la classe. Pour implémenter les méthodes, dont la syntaxe est similaire au C, on utilise l'opérateur de résolution de portée `Toto::` .

En Objective-C, les attributs et les méthodes ne peuvent être mélangés. Les attributs sont déclarés entre accolades, et les méthodes leur succèdent. Le corps des méthodes est spécifié dans une partie `@implementation`.

Il existe une différence fondamentale avec le C++, dans la mesure où des méthodes peuvent être implémentées sans avoir été déclarées dans l'interface. Ce point est détaillé plus tard. Sommairement, cela permet d'alléger les fichiers *.h* en ne répétant pas dans une sous-classe les déclarations des méthodes virtuelles redéfinies qui ne sont appelées qu'automatiquement, comme un destructeur par exemple. Voyez la section 4.3.2 page 22 pour plus d'explications.

Les méthodes sont précédées du signe « - » ou parfois du signe « + », pour différencier méthodes d'instance et méthodes de classe (cf. section 3.3.9 page 19). Ce signe n'a rien à voir avec la notation UML signifiant *public* ou *private*. Les types des paramètres sont encadrés de parenthèses, et surtout les paramètres sont séparés par des « : ». Voyez la section 3.3.1 page 13 pour plus de détails sur la syntaxe des prototypes.

Notez aussi qu'il n'y a pas de point-virgule nécessaire à la fin d'une déclaration de classe en Objective-C. Remarquez également que c'est bien `@interface` qui est employé pour déclarer une classe, et non `@class` comme on aurait pu le croire. Le mot-clef `@class` est utilisé dans les déclarations anticipées (*forward*) (cf. section 3.2.2 de la présente page). Sachez enfin que si la classe ne déclare pas de données d'instance, le bloc d'accolades ouvrantes-fermantes (qui serait vide) n'est pas nécessaire.

3.2.2 Déclarations forward : `@class`, `@protocol`

Pour éviter les dépendances cycliques des fichiers d'en-tête, il est nécessaire de recourir à la *déclaration forward* des classes lorsqu'on a juste besoin de spécifier leur existence, et non leur définition. En C++, le mot-clef `class` remplit aussi ce rôle ; en Objective-C, on utilise `@class`. Il existe aussi le mot-clef `@protocol` pour anticiper la déclaration des protocoles (cf. section 4.4 page 23).

C++

```
//Dans le fichier Toto.h

#ifndef __TOTO_H__
#define __TOTO_H__

class Titi; //déclaration forward
class Toto
{
  Titi* titi;

  public:
    void utiliserTiti(void);
};

#endif
```

```
//Dans le fichier Toto.cpp

#include "Toto.h"
#include "Titi.h"

void Toto::utiliserTiti(void)
{
 ...
}
```

Objective-C	

`//Dans le fichier Toto.h`
 	
`@class Titi; //déclaration forward`
`@interface Toto : NSObject`
`{`
`  Titi* titi;`
`}`
 	
`-(void) utiliserTiti;`
 	
`@end`
	`//Dans le fichier Toto.m`
	
	`#import "Toto.h"`
	`#import "Titi.h"`
	
	`@implementation Toto`
	
	`-(void) utiliserTiti`
	`{`
	`  ...`
	`}`
	
	`@end`

3.2.3 `public`, `private`, `protected`

Le modèle objet repose en partie sur l'encapsulation des données, qui permet de limiter leur visibilité dans différentes portions du code, pour offrir des garanties d'intégrité.

C++	Objective-C

`class Toto`
`{`
`  public:`
`    int x;`
`    int tata();`
 	
`  protected:`
`    int y;`
`    int titi();`
 	
`  private:`
`    int z;`
`    int tutu();`
`};`
	`@interface Toto : NSObject`
	`{`
	`  @public:`
	`    int x;`
	
	`  @protected:`
	`    int y;`
	
	`  @private:`
	`    int z;`
	`}`
	
	`-(int) tata;`
	`-(int) titi;`
	`-(int) tutu;`
	`@end`

En C++, attributs et méthodes peuvent appartenir à un contexte de visibilité `public`, `protected` ou `private`. Si elle n'est pas spécifiée, la visibilité est `private`.

En Objective-C, seules les données d'instance peuvent être `public`, `protected` ou `private`, et la visibilité par défaut est ici `protected`. Les méthodes ne peuvent être que publiques. On peut cependant imiter le mode privé, en implémentant des méthodes uniquement dans la partie `@implementation`, sans les déclarer dans la partie `@interface`, ou en utilisant la notion de catégories de classe (cf. section 4.5 page 26). Cela n'empêche pas les méthodes d'être appelées, mais elles sont moins visibles. Notez que le fait de pouvoir implémenter des méthodes sans les avoir déclarées dans l'interface est une fonctionnalité propre à l'Objective-C, et qui sert un autre but, expliqué en section 4.3.2 page 22.

On ne peut pas en Objective-C spécifier un caractère `public`, `protected` ou `private` pour l'héritage. ll est forcément `public`. L'héritage en Objective-C est beaucoup plus semblable à celui du Java que du C++ (section 4 page 22).

3.2.4 Attributs `static`

Il n'est pas possible en Objective-C de déclarer des données de classe, (attributs `static` en C++). On peut en revanche les obtenir par un moyen détourné. Il suffit pour cela d'utiliser une variable globale dans le fichier d'implémentation (éventuellement munie de l'attribut `static` du C limitant sa portée). La classe peut alors définir des accesseurs dessus (par des méthodes de classe ou des méthodes normales), et son initialisation peut être effectuée dans la méthode `initialize` de la classe (cf. section 5.1.10 page 35).

3.3 Méthodes

La syntaxe des méthodes en Objective-C est bien différente des fonctions C classiques. Cette section explique l'intérêt de cette syntaxe particulière, et déborde un peu sur le principe de l'envoi de messages, sur lequel repose Objective-C.

3.3.1 Prototype et appel, méthodes d'instance, méthodes de classe

- Une méthode est précédée d'un « − » si c'est une méthode d'instance (le cas le plus courant), et d'un « + » si c'est une méthode de classe (`static` en C++). Ce symbole n'a rien à voir avec la signification *public* ou *private* de l'UML. Rappelons que les méthodes sont toujours publiques en Objective-C ;
- les types de retour et des paramètres sont entre parenthèses ;
- les paramètres sont séparés par le caractère « : » ;
- les paramètres peuvent recevoir une **étiquette**, un nom avant le « : ». **L'étiquette fait alors partie du nom de la méthode et le modifie**. Cela rend les appels à cette fonction extrêmement lisibles. À l'usage, cette pratique est incontournable. Il est à noter que le premier paramètre ne peut recevoir d'étiquette autre que le nom de la méthode qui le précède ;
- le nom d'une méthode peut être strictement le même que celui d'un attribut, sans provoquer de conflit. Cela est très utile pour écrire des accesseurs en lecture (cf. section 6.4.8 page 48).

C++

```
//prototype
void Array::insertObject(void *anObject, unsigned int atIndex)

//utilisation avec une instance "etagere" de la classe Array
//et un objet "livre"
etagere.insertObject(livre, 2);
```

Objective-C

Sans étiquette (transcription directe d'un prototype C++)

```
//prototype
//La méthode se nomme "insertObject::", les deux-points servant à séparer
//les paramètres (il ne s'agit pas d'un opérateur de résolution de portée
//comme en C++)
-(void) insertObject:(id)anObject:(unsigned int)index

//utilisation avec une instance "etagere" de la classe Array
//et un objet "livre".
[etagere insertObject:livre:2];
```

Avec étiquette

```
//prototype. On assigne ici au paramètre "index" l'étiquette "atIndex".
//La méthode se nomme maintenant "insertObject:atIndex:"
//Cela permettra de lire cet appel comme une phrase
-(void) insertObject:(id)anObject atIndex:(unsigned int)index

//utilisation avec une instance "etagere" de la classe Array
//et un objet "livre"
[etagere insertObject:livre:2];         //Erreur !
[etagere insertObject:livre atIndex:2]; //OK
```

Notez que la syntaxe à base de crochets ne se lit pas comme *appeler la méthode* `insertObject` de l'objet « etagere » mais plutôt comme *envoyer le message* `insertObject` à l'objet « etagere ». Cette phrase traduit tout le dynamisme d'Objective-C. On peut envoyer les messages que l'on souhaite à une cible donnée. Si elle n'est pas capable de traiter ce message, elle l'ignorera (en pratique, une exception est levée, mais le programme ne s'interrompt pas). Si à un instant donné de l'exécution du programme la cible sait traiter le message, elle le fera avec la méthode correspondante. Il y a tout de même des avertissements de la part du compilateur si un message est envoyé à un objet de classe connue, pour laquelle on sait que le message est invalide ; mais ce n'est pas considéré comme une erreur (du fait du possible *forwarding* cf. section 3.4.3 page 20). Si la cible du message n'est connue que sous le type `id`, il n'y a que lors de l'exécution du programme qu'il pourra être déterminé si le message peut être traité. Aucun avertissement n'est donc levé lors de la compilation.

3.3.2 `this`, `self` et `super`

Il existe deux cibles particulières pour un message : `self` et `super`. `self` représente l'objet courant (équivalent de `this` en C++), `super` représente la classe mère, comme en Java. Le mot-clef `this` n'existe pas en Objective-C, `self` le remplace.

Remarque : `self` n'est cependant pas un véritable mot-clef, c'est un paramètre caché que reçoit toute méthode, et qui représente l'objet courant. On peut en changer la valeur, contrairement au `this` du C++, mais cette pratique n'est utilisée que dans les constructeurs (cf. section 5.1 page 29).

3.3.3 Accès aux données d'instance de son objet déclencheur

Tout comme en C++, une méthode en Objective-C peut accéder aux données d'instance de son objet déclencheur. L'éventuel `this->` du C++ est remplacé par `self->`.

C++	Objective-C

`class Toto`
`{`
`  int x;`
`  int y;`
`  void  f(void);`
`};`
 	
`void Toto::f(void)`
`{`
`  x = 1;`
`  int y; //crée une ambiguité avec`
`  //this->y`
`  y = 2; //utilise le y local`
`  this->y = 3; //résout l'ambiguité`
`}`
	`@interface Toto : NSObject`
	`{`
	`  int x;`
	`  int y;`
	`}`
	
	`-(void) f;`
	`@end`
	
	`@implementation Toto`
	`-(void) f`
	`{`
	`  x = 1;`
	`  int y;  //crée une ambiguité avec`
	`  //self->y`
	`  y = 2; //utilise le y local`
	`  self->y = 3; //résout l'ambiguité`
	`}`
	`@end`

3.3.4 Identifiant et signature du prototype, surcharge

Une fonction est une partie de code qui doit pouvoir être référencée, par exemple pour utiliser des pointeurs de fonctions, ou des foncteurs. De plus, si le nom de la fonction est un bon candidat pour jouer le rôle d'identifiant, il ne doit toutefois pas poser problème en cas de surcharge de ce nom. Les langages C++ et Objective-C sont antagonistes dans leur façon de différencier les prototypes. En effet, le premier se focalise sur les types des paramètres, et le second sur les étiquettes des paramètres.

En C++, deux fonctions de même nom peuvent être différenciées par le type des paramètres. Dans le cas des méthodes, l'option `const` est également déterminante.

C++

```
int f(int);
int f(float); //OK, float est différencié de int

class Toto
{
  public:
    int g(int);
    int g(float); //OK, float est différencié de int
    int g(float) const; //OK, le const est discriminant
};

class Titi
{
  public:
    int g(int); //OK, on est dans Titi::, différenciable de Toto::
}
```

En Objective-C, les fonctions sont celles du C : elles ne peuvent être surchargées (à moins d'utiliser un compilateur compatible avec le C99, comme l'est `gcc`). En revanche, les méthodes (qui ont une syntaxe différente) sont différenciables à travers les étiquettes des paramètres.

Objective-C

```
int f(int);
int f(float); //Erreur : fonctions C non surchargeables (sauf en C99)

@interface Toto : NSObject
{
}

-(int) g:(int) x;
-(int) g:(float) x; //Erreur : cette méthode n'est pas différenciée
                    // de la précédente (pas d'étiquette)
-(int) g:(int) x :(int) y;   //OK : il y a deux étiquettes anonymes
-(int) g:(int) x :(float) y; //Erreur : indifférenciable de la méthode
                             //précédente
-(int) g:(int) x andY:(int) y;   //OK : la deuxième étiquette est "andY"
-(int) g:(int) x andY:(float) y; //Erreur : indifférenciable de la
                                 //méthode précédente
-(int) g:(int) x andAlsoY:(int) y; //OK : la deuxième étiquette est
                                   //"andAlsoY", différent de "andY"
@end
```

Cette méthode de différenciation par les noms permet d'exprimer simplement quel est le nom « exact » de la fonction, comme expliqué ci-après.

```
@interface Titi : NSObject {}

//le nom de cette méthode est "g"
-(int) g;

//le nom de cette méthode est "g:"
-(int) g:(float) x;

//le nom de cette méthode est "g::"
-(int) g:(float) x :(float) y;

//le nom de cette méthode est "g:andY:"
-(int) g:(float) x andY:(float) y;

//le nom de cette méthode est "g:andZ:"
-(int) g:(float) x andZ:(float) z
@end
```

Comme on le voit, deux méthodes Objective-C ne sont pas différenciées par les types mais par les noms. C'est donc grâce à ce nom qu'on dispose de l'équivalent des « pointeurs de méthodes », appelés *sélecteurs*, détaillés dans la section 3.3.5 de la présente page.

3.3.5 Pointeur de méthode : Sélecteur

En Objective-C, seules les *méthodes* ont cette syntaxe particulière utilisant parenthèses et étiquettes. Il n'est pas possible de déclarer des *fonctions* avec cette syntaxe. La notion de pointeur de fonction est la même en C qu'en Objective-C. C'est en revanche pour les pointeurs de *méthodes* qu'un problème se pose. La notion de pointeur de méthode n'est pas implémentée en Objective-C comme en C++.

En C++, il s'agit d'une syntaxe difficile, mais cohérente avec le C : un pointeur qui se focalise sur les types.

C++

```
class Toto
{
  public:
    int f(float x) {...}
};

Toto titi
int (Toto::*p_f)(float) = &Toto::f; //Pointeur vers Toto::f

(titi.*p_f)(1.2345); //appel de titi.f(1.2345);
```

En Objective-C, un nouveau type a été introduit. Ce « pointeur de méthode » s'appelle un *sélecteur*. Il est de type `SEL` et sa valeur est calculée par un appel à `@selector` sur le nom exact de la méthode (comprenant les étiquettes des paramètres). Déclencher la méthode peut se faire grâce à la classe `NSInvocation`. Dans la plupart des cas, les méthodes utilitaires de la famille `performSelector` (héritées de NSObject) sont plus pratiques, mais plus limitées. Les trois méthodes `performSelector`: les plus simples sont les suivantes :

```
-(id) performSelector:(SEL)unSelecteur;
-(id) performSelector:(SEL)unSelecteur withObject:(id)unObjetPourParametre;
-(id) performSelector:(SEL)unSelecteur withObject:(id)unObjetPourParametre
                                       withObject:(id)unAutreObjetPourParametre;
```

La valeur renvoyée est celle de la méthode déclenchée. Pour des méthodes dont les paramètres ne sont pas des objets, il faudra songer à utiliser des classes d'encapsulation comme `NSNumber`, pouvant fournir des `float`, des `int`, etc. implicitement. On peut aussi se tourner vers la classe `NSInvocation`, plus générale et plus puissante (voir la documentation).

D'après ce qui précède, rien n'empêche d'essayer de déclencher une méthode dans un objet dont la classe ne l'implémente pas. Mais le déclenchement ne découle que de l'acceptation du message. Une exception, interceptable, est levée dans le cas où la méthode à déclencher n'est pas connue par l'objet, mais le programme ne s'interrompt pas brutalement. En outre, on peut tester explicitement si un objet sait traiter une méthode *via* un appel à `respondsToSelector`:.

Enfin, la valeur de `@selector()` étant évaluée à la compilation, elle ne pénalise pas le temps d'exécution.

Objective-C

```
@interface Larbin : NSObject {}
  -(void) lireDossier:(Dossier*)dossier;
@end

//Supposons un tableau tab[] de 10 larbins et un dossier "dossier"

//utilisation simple
for(i=0 ; i<10 ; ++i)
  [tab[i] lireDossier:dossier];

//simplement pour l'exemple, utilisation de performSelector:
for(i=0 ; i<10 ; ++i)
  [tab[i] performSelector:@selector(lireDossier:) withObject:dossier];

//le type d'un sélecteur est SEL. La version ci-dessous n'est pas plus
//efficace que la précédente, car @selector() est évalué à la compilation
SEL selecteur = @selector(lireDossier:);
for(i=0 ; i<10 ; ++i)
  [tab[i] performSelector:selecteur withObject:dossier];

//sur un objet "toto" de type inconnu (id), le test n'est pas
//obligatoire mais empêche la levée  d'exception si l'objet
//n'a pas de méthode lireDossier:.
if ([toto respondsToSelector:@selector(lireDossier:)])
  [toto performSelector:@selector(lireDossier:) withObject:dossier];
```

Un intérêt du sélecteur est de pouvoir être employé comme paramètre de fonction de façon extrêmement simple. Des algorithmes génériques, comme le tri, peuvent ainsi être aisément spécialisés (cf. 11.3 page 58).

Un sélecteur n'est cependant pas à proprement parler un pointeur de fonction ; son type sous-jacent réel est une chaîne de caractères, enregistrée auprès du *run-time*, identifiant une méthode. Lorsqu'une classe est chargée en mémoire, ses méthodes sont ainsi inscrites dans une table pour que fonctionne le mécanisme de `@selector()`. De cette façon, par exemple, l'égalité de deux sélecteurs est une égalité d'adresses, et non une égalité de chaîne de caractères ; l'opérateur `==` peut donc être employé.

L'adresse réelle d'une méthode, en tant que fonction C, est obtenue par un mécanisme différent, présenté brièvement en section 11.3.2 page 58, avec le type `IMP`. Son utilisation est plus rare, et surtout liée à l'optimisation. La résolution des appels virtuels, par exemple, est gérée par les sélecteurs, mais pas par les `IMP`. L'équivalent Objective-C des pointeurs de méthodes du C++ réside donc bien dans les sélecteurs.

Notez enfin que si `self` est en Objective-C, à l'instar de `this` en C++, un paramètre caché de chaque méthode représentant l'objet en cours, il en existe un deuxième, `_cmd`, qui est le sélecteur de la méthode courante.

Objective-C

```
@implementation Toto

-(void) f:(id)parametre //équivalent à une fonction C de type
                        //"void f(id self, SEL _cmd, id parametre)"
{
  id objetCourant = self;
  SEL methodeCourante = _cmd;
  [objetCourant performSelector:methodeCourante
                     withObject:parametre]; //appel récursif
  [self performSelector:_cmd withObject:parametre]; //idem
}

@end
```

3.3.6 Paramètres par défaut

L'Objective-C ne permet pas de spécifier des valeur par défaut pour les paramètres. Il faut donc créer autant de fonctions que nécessaire lorsque plusieurs nombres d'arguments sont envisageables. Dans le cas des constructeurs, il faut se référer à la notion d'*initialisateur désigné* (section 5.1.7 page 33) pour respecter les canons.

3.3.7 Nombre d'arguments variable

L'Objective-C permet de spécifier des méthodes au nombre d'arguments variable. Tout comme en C, il s'agit de rajouter « ... » en dernier argument. C'est une pratique fort rare pour l'utilisateur, même si de nombreuses méthodes implémentées dans Cocoa le font. Le lecteur curieux peut consulter la documentation Objective-C pour de plus amples détails sur ce point.

3.3.8 Arguments muets

En C++, il est possible de ne pas nommer tous les paramètres dans le prototype d'une fonction, puisque leur type suffit à caractériser la signature. En Objective-C, cela n'est pas possible.

3.3.9 Modificateurs de prototype (`const`, `static`, `virtual`, « `= 0` », `friend`, `throw`)

En C++, un certain nombre de modificateurs peuvent être ajoutés aux prototypes des fonctions. Aucun d'eux n'existe en Objective-C. En voici la liste :

- `const` : une méthode ne peut être spécifiée `const`. Le mot-clef `mutable` n'existe donc pas ;
- `static` : la différence entre méthode d'instance et méthode de classe se fait par l'utilisation du « - » ou du « + » devant le prototype ;
- `virtual` : toutes les méthodes sont virtuelles en Objective-C, ce mot-clef est donc inutile. Les fonctions virtuelles **pures** s'implémentent par un protocole formel (cf. section 4.4 page 23) ;
- `friend` : il n'y a pas de notion de classe ou méthode amie en Objective-C ;
- `throw` : en C++, on peut autoriser une méthode à ne transmettre que certaines exceptions. En Objective-C, ce n'est pas le cas.

3.4 Messages et transmission

3.4.1 Envoi d'un message à `nil`

Par défaut, il est légal d'envoyer un message (appeler une méthode) sur `nil`. Le message est simplement ignoré. Cela permet de simplifier le code en diminuant fortement le nombre de tests nécessaires avec le pointeur nul. GCC dispose d'une option pour désactiver ce comportement pratique, à des fins d'optimisation.

3.4.2 Délégation d'un message vers un objet inconnu

La délégation est courante avec certains éléments d'interface graphiques de Cocoa (tableaux, arborescence...). Il s'agit d'exploiter le fait qu'il est possible d'envoyer des messages à un objet inconnu. Un objet peut se décharger de certaines tâches en exploitant un objet assistant.

```
//Supposons l'existence d'une fonction d'attribution d'assistant
-(void) setStagiaire:(id)esclave
{
  [stagiaire autorelease]; //voir la section sur la gestion mémoire
  stagiaire = [esclave retain];
}
```

```
//la méthode faireUnTrucPenible peut implémenter une délégation
-(void) faireUnTrucPenible:(id) truc
{
  //le stagiaire est inconnu.
  //On vérifie qu'il peut traiter le message
  if ([stagiaire respondsToSelector:@(faireUnTrucPenible:)])
    [stagiaire faireUnTrucPenible:truc];
  else
    [self changerDeStagiaire];
}
```

3.4.3 Forwarding : gestion d'un message inconnu

En C++, on ne peut pas à la compilation forcer un objet à essayer d'exécuter une méthode qu'il n'implémente pas. En Objective-C, ce n'est pas pareil : on peut toujours envoyer un message à un objet. S'il ne peut pas le traiter, il l'ignorera (en levant une exception) ; ou mieux : plutôt que de l'ignorer, il peut le retransmettre à un tiers.

Notez que lorsque le compilateur détecte un envoi de message à un objet, et que d'après son type cet objet ne connaît pas le message, un avertissement est levé. Ce n'est cependant pas une erreur, car lorsqu'un objet reçoit un message qu'il ne sait pas traiter, une seconde chance lui est offerte. Cette seconde chance prend forme par un appel automatique à la méthode `forwardInvocation:`, qui peut être surchargée pour re-router le message au dernier moment. C'est bien sûr une méthode de `NSObject` qui par défaut ne fait rien. C'est encore une façon de gérer des objets assistants.

```
-(void) forwardInvocation:(NSInvocation*)anInvocation
{
  //si on est ici, c'est que l'objet ne sait pas traiter
  //le message de l'invocation
  //le selecteur fautif est accessible par l'envoi du message "selector"
  //à l'objet "anInvocation"
  if ([unAutreObjet respondsToSelector:[anInvocation selector]])
    [anInvocation invokeWithTarget:unAutreObjet];
  else  //ne pas oublier de tenter sa chance avec la superclasse
    [super forwardInvocation:anInvocation];
}
```

Remarquons toutefois que si un message peut être traité dans une `forwardInvocation:`, et uniquement là, un test d'aptitude basé sur un `respondsToSelector:` renverra `NO` malgré tout. En effet, le mécanisme de `respondsToSelector:` ne peut pas tester l'envoi de message en conditions réelles, et ne peut donc soupçonner la présence de toutes les roues de secours.

Le mécanisme de la *foward invocation* peut sembler une mauvaise pratique, car elle détourne un cas d'erreur pour exécuter du code. En réalité, de très bons usages peuvent en être fait, comme dans l'implémentation du `NSUndoManager` de Cocoa (gestionnaire de *undo/redo*), qui y gagne une syntaxe remarquablement efficace : le *UndoManager* peut enregistrer les appels de méthodes dont il n'est pas la cible réelle.

3.4.4 Downcasting

Le *downcasting* est nécessaire en C++ pour appeller les méthodes d'une classe dérivée, lorsque l'on ne dispose que du pointeur d'une classe mère. Cette pratique n'est pas incorrecte, *via* l'appel à `dynamic_cast`. En Objective-C, elle n'est pas incontournable, puisqu'un message peut être envoyé même si le type semble ne pas indiquer que l'objet puisse y répondre.

Dans le cas d'un type explicite, pour éviter un avertissement à la compilation, on peut *caster* le type de l'objet ; il n'y a pas d'opérateur explicite de *downcasting* en Objective-C, on utilise le *cast* traditionnel du C.

```
//NSMutableString est une sous-classe de NSString (chaîne de caractères)
//permettant des modifications de la chaîne.
//la méthode "appendString:" existe pour NSMutableString, pas pour NSString

NSMutableString* chaineModifiable = ...initialisation...
NSString* chaine = chaineModifiable;//stockage en pointeur de classe mère

//les codes ci-dessous sont corrects
[chaine                   appendString:@"foo"];//avertissement du compilateur
[(NSMutableString*)chaine appendString:@"foo"];//pas d'avertissement
[(id)chaine               appendString:@"foo"];//pas d'avertissement
```

4 Héritage

4.1 Héritage simple

Objective-C implémente bien sûr la notion d'héritage, mais ne supporte pas l'héritage multiple. Cette apparente limitation est en revanche complétée par d'autres concepts (protocoles, catégories de classe) qui sont expliqués plus loin dans ce document (sections 4.4 page suivante, 4.5 page 26).

C++	Objective-C

`class Toto : public Tutu,`
`protected Tata`
`{`
`}`
	`@interface Toto : Tutu //héritage simple`
	`//il y a des techniques pour`
	`//"dériver" aussi de Tata`
	`{`
	`}`
	`@end`

En C++, une classe peut dériver d'une ou plusieurs autres classes, de façon `public`, `protected` ou `private`. Dans les méthodes, on peut faire référence à une super-classe grâce à l'opérateur de résolution de portée (`Tutu`::, `Tata`::).

En Objective-C, on ne peut dériver que d'une seule classe, de façon publique. Une méthode peut faire référence à la classe mère comme en Java, par le (faux) mot-clef `super`.

4.2 Héritage multiple

Objective-C n'implémente pas l'héritage multiple, et le compense par d'autres concepts, les *protocoles* (cf. 4.4 page suivante) et les *catégories* (cf. 4.5 page 26).

4.3 Virtualité

4.3.1 Méthodes virtuelles

Les méthodes sont automatiquement et obligatoirement virtuelles en Objective-C. On n'a donc pas besoin de le spécifier. De ce fait, le mot-clef `virtual` n'existe pas et n'a pas d'équivalent.

4.3.2 Redéfinition silencieuse des méthodes virtuelles

Il est possible en Objective-C d'implémenter une méthode sans l'avoir déclarée au préalable dans l'interface. Cette fonctionnalité n'est pas destinée à compenser l'absence du qualificateur `@private` pour les méthodes : même si elle peut servir à « cacher »des méthodes, cela n'empêche pas de les appeler. Elle permet en revanche d'alléger considérablement les déclarations d'interface pour les méthodes héritées.

Ce n'est pas une mauvaise pratique : les méthodes concernées sont plutôt les méthodes « bien connues » déclarées dans les super-classes. De nombreuses méthodes de la classe racine `NSObject` sont ainsi redéfinies silencieusement la plupart du temps. On peut citer par exemple le constructeur `init` (cf. section 5.1 page 29), le destructeur `dealloc` (cf. section 5.2 page 35), la méthode de dessin `drawRect`: des vues, etc.

À l'usage, les interfaces sont plus simples, même si l'on perd de vue ce que l'on *peut* redéfinir, ce qui rend obligatoire la consultation régulière des documentations des classes mères.

Notez enfin que la notion de méthode virtuelle pure, dont la redéfinition est obligatoire dans les sous-classes, est résolue par le concept de *protocoles formels* (cf. section 4.4.1 page suivante page suivante).

4.3.3 Héritage virtuel

L'héritage virtuel n'a pas de raison d'être en Objective-C, puisque l'héritage y est uniquement simple et ne soulève aucun des problèmes de l'héritage multiple.

4.4 Protocoles

Java et C# compensent l'absence d'héritage multiple par la notion d'*interface*. En Objective-C, la même notion est utilisée, mais est appelée *protocole*. En C++, on utiliserait une classe abstraite. Un protocole n'est pas une classe à proprement parler, car il ne peut proposer que des méthodes, et ne peut contenir de données. Il existe deux types de protocoles : les protocoles *formels* et les protocoles *informels*.

4.4.1 Protocole formel

Un protocole formel est un ensemble de méthodes qui doivent être implémentées par toute classe adhérente. Cela peut aussi être vu comme une certification accordée à une classe lorsqu'elle implémente tout ce qui est nécessaire à un service donné. Une classe peut adhérer à un nombre quelconque de protocoles.

C++

```
class MouseListener
{
  public:
    virtual bool mousePressed(void) = 0; //méthode virtuelle pure
    virtual bool mouseClicked(void) = 0; //méthode virtuelle pure
};

class KeyboardListener
{
  public:
    virtual bool keyPressed(void) = 0; //méthode virtuelle pure
};

class Toto : public MouseListener,  public KeyboardListener {...}

//Toto DOIT implémenter mousePressed, mouseClicked et keyPressed
//On pourra donc l'utiliser comme auditeur d'événements
```

Objective-C

```
@protocol MouseListener
  -(BOOL) mousePressed;
  -(BOOL) mouseClicked;
@end

@protocol KeyboardListener
  -(BOOL) keyPressed;
@end

@interface Toto : NSObject  <MouseListener,  KeyboardListener>
{
...
}
@end

//Toto DOIT implémenter mousePressed, mouseClicked et keyPressed
//On pourra donc l'utiliser comme auditeur d'événements
```

En C++, un protocole s'implémente par une classe abstraite et des méthodes virtuelles pures. La classe abstraite du C++ est cependant plus puissante que le protocole d'Objective-C, car elle peut contenir des attributs.

En Objective-C, le protocole est un concept spécifique. La syntaxe à base de chevrons <...> n'a rien à voir avec les templates C++, qui n'existent pas en Objective-C.

Notez que l'on peut implémenter toutes les méthodes d'un protocole dans une classe, sans pour autant indiquer explicitement dans le code qu'elle y adhère. L'inconvénient est que la méthode `conformsToProtocol`: renvoie alors `NO`. Pour des raisons d'efficacité, cette fonction ne teste pas la conformité à un protocole méthode par méthode, mais se base sur la conformité explicite donnée par le développeur. Dans un tel cas, la réponse négative de `conformsToProtocol`: n'empêche pas le programme de se comporter correctement par ailleurs. Voici le prototype de la méthode `conformsToProtocol`: :

```
-(BOOL) conformsToProtocol:(Protocol*)protocol
//un objet Protocol est obtenu par un appel à @protocol(nom du protocole)
```

Au type d'un objet conforme à un protocole formel, on peut ajouter le nom du protocole lui-même, entre chevrons. Cela sert à réaliser des assertions. Par exemple :

```
//la méthode standard de Cocoa qui suit prend un paramètre de type
//quelconque (id), mais qui doit respecter le protocole NSDraggingInfo
-(NSDragOperation) draggingEntered:(id <NSDraggingInfo>)sender;
```

4.4.2 Méthodes optionnelles

Parfois, il est souhaitable qu'une classe adhère à un protocole, pour montrer qu'elle est candidate à une certaine tâche, mais sans pour autant l'obliger à implémenter *toutes* les méthodes du protocole. Par exemple, en Cocoa, la notion d'objet *délégué* est très présente : un objet peut se voir attribuer un assistant, auquel il peut déléguer certains travaux, mais pas forcément tous.

Une solution immédiate consiste à scinder un protocole formel en plusieurs protocoles formels, puis à n'adhérer qu'à un sous-ensemble de ces protocoles, mais cela deviendrait vite laborieux et serait peu pratique.

Avec Objective-C 1.0, les *procotoles informels* pouvaient être utilisés (cf. section 4.4.3). Avec Objective-C 2.0, les nouveaux mots-clefs `@optional` et `@required` permettent de distinguer des méthodes optionnelles des méthodes obligatoires.

```
@protocol Stagiaire

@required //partie obligatoire
-(void) faireLeCafe;
-(void) faireDesPhotocopies:(Document*)document exemplaires:(int)exemplaires;

@optional //partie optionnelle
-(void) faireLeMénage;

@required //on peut répéter des sections required/optional
-(void) apporterLeCafé;

@end
```

4.4.3 Protocole informel

Le protocole informel n'a de « protocole » que le nom, au sens où il n'est pas un outil de contrainte sur le code. Il est en revanche « informel » par sa nature qui le voue à l'auto-documentation du code.

Un protocole informel sert à regrouper des méthodes du même champ d'applications, permettant au développeur d'utiliser, d'une classe à l'autre, un nom de groupe cohérent.

Il n'est donc pas si suprenant que ce ne soit pas avec une relaxation de protocole formel que l'on spécifie un protocole informel. On utilise un autre concept : la *catégorie de classe* (cf. section 4.5 page suivante).

Supposons un service « traiter des dossiers ». Le problème est la présence de dossiers verts, bleus et rouges. Après tout, si une classe `Larbin` ne peut traiter que les dossiers bleus, elle peut quand même rendre service. Il ne serait pas pratique d'utiliser trois protocoles formels `TraiterDossierVert`, `TraiterDossierBleu` et `TraiterDossierRouge`. On ajoute plutôt à la classe `Larbin` une *catégorie* `TraiterDossier`, contenant les méthodes de traitement de dossier qu'elle est capable d'accomplir. Cela s'écrit alors ainsi, en spécifiant entre parenthèses le nom que l'on choisit pour la catégorie (expliquée plus précisément en section 4.5 page suivante) :

```
@interface Larbin (TraiterDossier)
-(void) lireDossierBleu:(DossierBleu*) dossier;
-(void) jeterDossierBleu:(DossierBleu*) dossier;
@end
```

On peut imaginer d'autres classes utilisant la catégorie `TraiterDossier`, et proposant d'autres méthodes en rapport avec le service.

```
@interface LarbinConsciencieux (TraiterDossier)
-(void) traiterDossierBleu:(DossierBleu*) dossier;
-(void) traiterDossierRouge:(DossierRouge*) dossier;
@end
```

Un développeur tiers parcourant le code peut constater facilement que la classe a une catégorie `TraiterDossier`, il peut donc immédiatement la supposer candidate à certaines tâches, et n'a plus qu'à vérifier lesquelles exactement. S'il ne vérifie pas dans le code source, il peut toujours le faire à l'exécution :

```
if ([monLarbin respondsToSelector:@selector(traiterDossierBleu:)])
  [monLarbin traiterDossierBleu:dossier];
```

En vérité, à part pour la connaissance des prototypes, le protocole informel n'a pas d'utilité pour le compilateur, il ne restreint pas l'utilisation des objets. En revanche, il est précieux comme autodocumentation du code, rendant plus aisée l'utilisation d'une bibliothèque développée par un tiers.

4.4.4 Objet de type `Protocol`

À l'exécution du programme, un protocole est, comme une classe, représenté par un objet, en l'occurence du type `Protocol*`. Un tel objet peut servir de paramètre à une méthode telle `conformsToProtocol :` (cf. section 13.1.2 page 65).

Le mot-clef `@protocol`, servant à déclarer un protocole, donne aussi accès à un objet `Protocol*` à partir de son nom :
`Protocol* monProtocole = @protocol(`*`nom du protocole`*`)`.

4.4.5 Qualificateurs pour messages entre objets distants

Le dynamisme d'Objective-C permet à des objets distants de communiquer entre eux. Ils peuvent appartenir à des programmes distincts, sur des machines différentes, mais sont capables de se déléguer des tâches et de s'échanger des informations. Or, les protocoles formels sont un moyen idéal pour certifier des objets conformes à un service donné, quelle que soit leur origine. Le

concept de protocole formel a donc été enrichi de mots-clefs supplémentaires pour permettre une implémentation plus efficace des envois de message à distance.

Ces mots-clefs sont `in`, `out`, `inout`, `bycopy`, `byref` et `oneway`. Ils ne s'appliquent qu'aux objets distribués, et hors de la définition d'un protocole, ils ne sont pas réservés par le langage et peuvent être utilisés librement.

Ces mots-clefs sont insérés dans les prototypes des méthodes déclarées par un protocole formel pour en préciser le comportement. On peut ainsi savoir si les paramètres correspondent à des données d'entrée, de sortie ; on connaît leur mode de passage (par copie ou par référence) ; on sait si la méthode est synchrone ou non.

Les significations sont les suivantes :

- un paramètre spécifié `in` est une variable d'entrée ;
- un paramètre spécifié `out` est une variable de sortie ;
- un paramètre spécifié `inout` peut être utilisé comme entrée et sortie ;
- un paramètre spécifié `bycopy` est passé par copie ;
- un paramètre spécifié `byref` est passé par référence (sans copie) ;
- une méthode spécifiée `oneway` est asynchrone (on n'attend pas de retour immédiat), et a forcément `void` comme type de retour.

Voici par exemple une méthode asynchrone qui renvoie un objet :

```
-(oneway void) donneMoiUnObjetQuandTuPeux:(bycopy out id *) unObjet;
```

Par défaut, les paramètres sont considérés `inout`, sauf les pointeurs `const`, qui sont considérés `in`. Réduire le sens de `inout` à `in` ou `out` est une optimisation. Le mode de passage par défaut des paramètres est `byref`, et le comportement par défaut des méthodes est synchrone (sans `oneway`).

Pour les arguments passés par valeur, comme des variables non-pointeurs, `out` et `inout` ne signifient rien, seul `in` est correct.

4.5 Catégories de classe

Implémenter des *catégories* pour une classe permet de morceler sa définition et son implémentation. Chaque *catégorie* est un élément constituant de la classe. Une classe peut être implémentée par un nombre quelconque de catégories, mais elles ne peuvent déclarer de nouvelles données d'instance. On bénéficie alors des avantages suivants :

- Pour le développeur pointilleux, cela permet de regrouper des méthodes. Pour les classes très riches, cela permet d'isoler les différents rôles ;
- En conséquence immédiate, on bénéficie à la fois d'une compilation séparée, et de possibilités de travailler sur la même classe à plusieurs ;
- Si une interface de catégorie et son implémentation sont dans un fichier d'implémentation quelconque (fichier *.m*), cela revient à définir des méthodes *privées*, visibles uniquement dans ce fichier (mais exploitables malgré tout par quelqu'un de bien informé : il n'y a pas de restriction d'appel). Un nom de catégorie adapté pourrait être *TotoPrivateAPI* ;
- Cela permet également de personnaliser une classe différemment pour plusieurs applications, sans avoir à dupliquer le code source commun. N'importe quelle classe peut ainsi être étendue, même celles de Cocoa.

Le dernier point est important : en effet, pour nos besoins particuliers, de petites méthodes supplémentaires dans les classes standards seraient parfois agréables. Ce n'est pas un problème en soi, puisqu'il suffit d'une dérivation pour étendre les fonctionnalités d'une classe. Cependant, dans un contexte d'héritage simple, cela pourrait générer une arborescence de classe trop touffue. De plus, il peut être laborieux de créer, par exemple, une nouvelle classe `MyString` juste pour une méthode, et de devoir l'utiliser ensuite dans tout le programme. Les catégories de classe permettent de résoudre élégamment ce problème.

C++

```
class MyString : public string
{
  public:
    int compterVoyelles(void); //compte les voyelles
};

int MyString::compterVoyelles(void)
{
...
}
```

Objective-C

```
@interface NSString (QuiCompteLesVoyelles)
//Notez l'absence de {}
-(int) compterVoyelles; //compter les voyelles
@end

@implementation NSString (QuiCompteLesVoyelles)
-(int) compterVoyelles
{
...
}
@end
```

En C++, la nouvelle sous-classe est utilisable sans restrictions.

En Objective-C, la classe NSString (c'est une classe de Cocoa) se voit attribuer une extension valable dans l'intégralité du programme. Aucune nouvelle classe n'est créée. Tout objet NSString bénéficie de l'extension (même les chaînes constantes, cf. section 9.1 page 55). Attention : lors de la déclaration d'une catégorie de classe, aucune variable d'instance ne peut être ajoutée. Il n'y a donc pas de bloc *{...}*

Une catégorie peut même ne pas avoir de nom, ce qui est idéal pour une catégorie « privée ».

```
@interface NSString ()
//Notez l'absence de {}
-(int) maMethodePrivee;
@end

@implementation NSString ()
-(int) maMethodePrivee
{
...
}
@end
```

4.6 Utilisation conjointe de protocoles, catégories, dérivation

La seule restriction dans l'utilisation conjointe de protocoles, catégories et dérivation et de ne pas pouvoir en même temps déclarer une sous-classe et implémenter une catégorie ; cela nécessite deux étapes.

```
@interface Toto1 : SuperClasse <Protocole1, Protocole2, ... > //ok
@end

@interface Toto2 (Categorie)  <Protocole1, Protocole2, ... > //ok
@end

//ci-dessous : erreur de compilation
@interface Toto3 (Categorie)  : SuperClasse <Protocole1, Protocole2, ... >
@end

//pour bien faire :
@interface Toto3 : SuperClasse <Protocole1, Protocole2, ... > //étape 1
@end
@interface Toto3 (Categorie) //étape 2
@end
```

5 Instanciation

L'instanciation d'une classe soulève deux problèmes : comment est implémentée la notion de constructeur/destructeur/opérateur de copie, et comment est gérée la mémoire ?

D'abord une remarque très importante : en C et en C++, les variables sont dites « automatiques » par défaut : à moins d'être déclarées `static`, elles n'existent que dans leur bloc de définition. Seule la mémoire allouée dynamiquement persiste jusqu'au `free()` ou au `delete` adapté. Les objets n'échappent pas à cette règle en C++.

Cependant, en Objective-C, **tout objet est créé dynamiquement**. Cet état de fait est somme toute logique, le C++ étant un langage typé statiquement, et l'Objective-C étant dynamique. Le dynamisme serait entravé si les objets n'étaient pas créés lors de l'exécution du programme.

Voyez la section 6 page 40 pour des explications plus poussées sur la façon de maintenir ou libérer les objets.

5.1 Constructeurs, initialisateurs

5.1.1 Distinction entre *allocation* et *initialisation*

En C++, l'allocation et l'initialisation d'un objet sont confondues dans l'appel au constructeur. En Objective-C, ce sont deux méthodes différentes.

L'allocation est assurée par la **méthode de classe** `alloc`, qui a également pour effet d'initialiser toutes les données d'instance. Les données d'instance sont initialisées à 0, excepté le pointeur `isa` (est-un) de `NSObject`, initialisé de telle sorte qu'il décrive le type exact de l'objet lors de l'exécution du code. Si les données d'instance doivent être initialisées à des valeurs particulières, dépendantes des paramètres de construction, le code correspondant est déporté dans une **méthode d'instance**, dont le nom commence traditionnellement par *init*. La construction est ainsi clairement séparée en deux étapes : l'allocation et l'initialisation. Notez que le message `alloc` est envoyée à la *classe*, et le message `init...` est envoyé à l'*objet* instancié par `alloc`.

La phase d'initialisation n'est pas optionnelle, et un `alloc` devrait toujours être suivi d'un `init`, lequel, par le jeu des appels aux constructeurs des super-classes, doit finir par déclencher le `init` de `NSObject` assurant différentes tâches importantes.

En C++, le nom du constructeur est imposé par le langage. En Objective-C, ce n'est pas le cas, car à part le préfixe `init`, qui est traditionnel sans être obligatoire, le nom de la méthode est libre. Il est cependant très déconseillé de ne pas respecter ce canon, à tel point qu'il vaut mieux l'énoncer comme une loi : **le nom d'une méthode d'initialisation doit commencer par « init »**.

5.1.2 Utilisation de `alloc` et `init`

L'appel à `alloc` renvoie l'objet nouvellement créé sur lequel on effectue le `init`. L'appel à `init` renvoie aussi un objet. Dans la plupart des cas, ce sera l'objet initial. Mais parfois, comme par exemple si on utilise un singleton (objet dont on ne veut pas plus d'une instance à la fois), le `init` peut se permettre de substituer une autre valeur de retour. Il ne faut donc pas ignorer la valeur de retour d'un `init` ! Généralement, on enchaîne donc les appels à `alloc` et `init` sur la même ligne d'instructions.

C++

```
Toto* toto = new Toto;
```

Objective-C

```
Toto* toto1 = [Toto alloc];
[toto1 init]; //mauvais, on doit se soucier de la valeur de retour

Toto* toto2 = [Toto alloc];
toto2 = [toto2 init]; //ok, mais pas pratique

Toto* toto3 = [[Toto alloc] init]; //ok, c'est ainsi que l'on fait
```

Notez que pour savoir si un objet a été effectivement créé, C++ nécessite soit une interception d'exception, soit un test avec un 0 si `new(nothrow)` a été utilisé. Avec Objective-C, il suffit de tester si l'objet est à `nil`.

5.1.3 Exemple d'initialisateur correct

Les contraintes d'un initialisateur sont donc :
- d'avoir un nom commençant par *init* ;
- de renvoyer l'objet initialisé à utiliser ;
- d'appeler un `init` de la classe mère, de telle sorte qu'au moins le `init` de `NSObject` soit appelé ;
- de prendre en compte la valeur renvoyée par `[super init...]` ;
- de gérer proprement les erreurs de construction, volontaires ou héritées.

Ci-après est donné un code d'exemple de construction d'objet en C++ et en Objective-C. La gestion d'erreur est présentée dans la suite.

C++

```
class Point2D {
  public:
    Point2D(int x, int y);
  private:
    int x;
    int y;
};
Point2D::Point2D(int unX, int unY) {x = unX; y = unY;}
...
Point2D  p1(3,4);
Point2D* p2 = new Point2D(5, 6);
```

Objective-C

```
@interface Point2D : NSObject {
    int x;
    int y;
}
//Remarque : "id" est un peu l'équivalent de "void*" en Objective-C;
//c'est est le type "général" d'un objet.
-(id) initWithX:(int)unX andY:(int)unY;
@end

@implementation Point2D

-(id) initWithX:(int)unX andY:(int)unY
{
  //il faut penser à appeler un constructeur de la superclasse
  if (!(self = [super init])) //pour NSObject, c'est simplement init.
    return nil; //si la construction de la super-classe échoue, "nil" est renvoyé
  //et on renvoie nil à nouveau pour propager cette erreur

  self->x = unX;//en cas de succès, faire les initialisations supplémentaires
  self->y = unY;
  return self; //renvoyer l'objet lui-même
}
@end
...
Point2D* p1 = [[Point2D alloc] initWithX:3 andY:4];
```

5.1.4 `self = [super init...]`

La syntaxe la plus surprenante de la construction est `self = [super init...]`. Rappelons que `self` est un argument caché passé à tout méthode, et représentant l'objet courant. Il s'agit donc d'une variable locale, alors à quoi bon changer sa valeur ? Will Shipley[9] a tenté de montrer dans un document très intéressant que cette pratique était inutile. Sa démonstration reposait sur des hypothèses quant au *run-time* Objective-C qui se sont avérées fausses ; en réalité, il faut bien modifier la valeur de `self`, mais des explications ne sont pas superflues.

Il peut arriver que `[super init]` renvoie un autre objet que l'objet courant. Le singleton en serait un cas d'application, mais c'est un contre-argument de [9] : il est illogique d'appeler `init` une deuxième fois pour une classe singleton ; en arriver à ce stade traduit une erreur de conception en amont.

En revanche, une API peut très bien substituer, à un objet que l'on initialise, un autre objet. Core Data [1] fait cela, pour s'adapter au traitement spécial des données membres de l'objet, qui sont alors reliées à une base de données. Si l'on dérive la classe `NSManagedObject` fournie par Cocoa, il est indispensable de prendre garde à cette substitution.

Dans ce cas, `self` prendra successivement deux valeurs différentes : la première est celle renvoyée par `alloc`, la seconde est celle que renvoie `[super init...]`. Modifier la valeur de `self` n'est pas anodin : tout accès à une donnée membre l'utilise implicitement, comme le montre le code ci-dessous.

```
@interface B : A
{
int i;
}
@end

@implementation B

-(id) init
{
  //à ce stade, la valeur de self est celle qu'a renvoyé alloc

  //supposons que A fasse une substitution et renvoie un "self" différent
  id nouveauSelf = [super init];
  NSLog(@"%d", i);//affiche la valeur de self->i
  self = nouveauSelf; //on n'a pas l'impression de toucher à "i", et pourtant...
  NSLog(@"%d", i);//affiche la valeur de self->i, donc nouveauSelf->i,
                  //pas nécessairement la même que précedemment !
  return self;
}

@end

...
B* b = [[B alloc] init];
```

La forme concise `self = [super init]` est la plus simple pour ne pas faire d'erreurs par la suite. Cependant, il est légitime de se demander ce que devient l'objet pointé par l'« ancien » `self` : il doit effectivement être détruit.

La première consigne est simple : c'est le responsable de la substitution qui se charge de l'ancien `self`, donc c'est le `[super init...]` qui doit le désallouer. Par exemple, si vous *dérivez* de `NSManagedObject` (une classe Cocoa réalisant une substitution), alors vous n'avez pas à vous soucier de l'ancien `self`. En revanche, le développeur de `NSManagedObject`, lui, l'a pris en compte.

1. Core Data est une API Cocoa fournie par Apple

Si vous êtes amené à créer une classe réalisant une substitution, vous devez donc savoir comment désallouer un objet lors de sa construction. Ce problème est identique à celui de la gestion d'erreur : que faire si l'on veut faire échouer la construction d'un objet (paramètres invalides, ressources indisponibles...) ? Cette question est traitée dans la section 5.1.5.

5.1.5 Échec de l'initialisation

Lors de la construction d'un objet (l'initialisation en réalité), une erreur peut survenir, ou être déclenchée, à trois endroits différents :

- 1 : avant l'appel à `[super init...]` : si les paramètres de contruction sont considérés comment invalides, on peut interrompre l'initialisation au plus tôt ;
- 2 : au retour de `[super init...]` : si la super-classe échoue, il faut s'interrompre ;
- 3 : après l'appel à `[super init...]` : si une allocation de ressource supplémentaire échoue, par exemple.

Dans tous les cas, il faut renvoyer `nil`, et c'est le responsable de l'erreur qui doit désallouer l'objet courant. Ici, on est responsable de l'erreur dans les cas 1 et 3, mais pas dans le cas 2. Pour désallouer l'objet courant, il suffit d'utiliser `[self release]`, ce qui est le plus naturel (cf. section 6 page 40 sur la gestion mémoire, qui explique le `release`).

La destruction d'un l'objet déclenche la méthode `dealloc` (cf section 5.2 page 35 sur les destructeurs), il faut donc que cette destruction soit implémentée pour fonctionner même sur un objet partiellement initialisé. Le fait que les données d'instance soient initialisées à 0 par `alloc` est généralement assez pratique de ce point de vue.

```
@interface A : NSObject {
  unsigned int n;
}
-(id) initWithN:(unsigned int)valeur;
@end

@implementation A

-(id) initWithN:(unsigned int)valeur
{
  //cas n°1 (la construction est-elle raisonnable ?)
  if (valeur == 0) //ici, on veut une valeur strictement positive
  {
    [self release];
    return nil;
  }

  //cas n°2 (la super-classe fonctionne-t-elle ?)
  if (!(self = [super init])) //même si self est substitué, c'est la super-classe
    return nil; //qui est responsable, si erreur, de la désallocation des "self"

  //cas n°3 (l'initialisation peut-elle être complète ?)
  n = (int)log(valeur);
  void* p = malloc(n); //essai d'allocation de ressource...
  if (!p) //si cela échoue, nous voulons que ce soit une erreur
  {
    [self release];
    return nil;
  }
}

@end
```

5.1.6 Construction « éclatée » en `alloc`+`init`

Le principe de l'application successive de `alloc` puis `init` peut sembler laborieux dans certains cas. Il peut fort heureusement être court-circuité par ce que l'on appelle un *constructeur de commodité*. Expliquer ce qu'est un tel constructeur nécessite des connaissances sur la gestion mémoire en Objective-C. Les véritables explications sont donc déportées plus loin en section 6.4.6 page 43. Brièvement, un tel constructeur, dont le nom devrait toujours être préfixé par celui de la classe, a le même rôle qu'une méthode d'`init`, mais elle fait le `alloc` elle-même. Cependant, l'objet renvoyé est inscrit au bassin de libération programmée (*autorelease pool*, cf. section 6.4 page 41) et sera donc temporaire s'il ne lui est pas envoyé un `retain`. Un exemple d'utilisation est donné ci-après :

```
//laborieux
NSNumber* tmp1 = [[NSNumber alloc] initWithFloat:0.0f];
...
[tmp1 release];

//plus sympathique
NSNumber* tmp2 = [NSNumber numberWithFloat:0.0f];
...
//pas besoin de release
```

5.1.7 Constructeur par défaut : initialisateur désigné

La notion de constructeur par défaut n'a pas vraiment de sens en Objective-C. Les objets étant tous alloués dynamiquement, leur construction est toujours explicite. Cependant, on retrouve la problématique d'un initialisateur privilégié pour éviter une redondance de code malvenue. En effet, un initialisateur correct contient la plupart du temps un code similaire à :

```
if (!(self = [super init])) //"init" ou un autre initialisateur
  return nil;      // plus approprié de la superclasse

//en cas de succès...
//...ajouter du code...
return self;
```

Toute redondance de code étant à proscrire, il semble inopportun de reproduire ce schéma dans chaque initialisateur. La meilleure solution consiste effectivement à privilégier l'initialisateur le plus essentiel pour y placer ce code. Les autres initialisateurs ne feront qu'appeler cet initialisateur « de base », nommé *initialisateur désigné* (*designated initializer*). Logiquement, l'initialisateur désigné est celui qui a le plus de paramètres, puisqu'il est impossible en Objective-C de donner des valeurs par défaut aux paramètres.

```
-(id) initWithX:(int)x
{
  return [self initWithX:x andY:0 andZ:0];
}

-(id) initWithX:(int)x andY:(int)y
{
  return [self initWithX:x andY:y andZ:0];
}

//initialisateur désigné
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (!(self = [super init]))
    return nil;
  self->x = x;
  self->y = y;
  self->z = z;
  return self;
}
```

Si l'initialisateur désigné n'est pas celui qui a le plus de paramètres, ce n'est pas très pratique :

```
//Le code suivant n'est pas bien pratique.
-(id) initWithX:(int)x //initialisateur désigné
{
  if (!(self = [super init]))
    return nil;
  self->x = x;
  return self;
}
-(id) initWithX:(int)x andY:(int)y
{
  if (![self initWithX:x])
    return nil;
  self->y = y;
  return self;
}
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (![self initWithX:x])
    return nil;
  self->y = y;
  self->z = z;
  return self;
}
```

5.1.8 Listes d'initialisation et valeur par défaut des données d'instance

Les *listes d'initialisation* des constructeur C++ n'existent pas en Objective-C. Mais il est important de noter que contrairement au C++, en Objective-C, **les bits des données d'instance sont tous initialisés à 0** par `alloc`. Tous les types de base prennent donc la valeur 0 par défaut, ainsi que les pointeurs (qui sont donc à `nil`). Rappelons que cela ne pose pas de problèmes pour les éventuelles données d'instance qui seraient des objets, car elles ne pourraient être en réalité que des pointeurs d'objets : en Objective-C, les objets ne peuvent être alloués que dynamiquement.

5.1.9 Constructeur virtuel

Il est possible en Objective-C d'obtenir de véritables constructeurs virtuels. Voyez pour cela la section 6.4.6 page 43, présentée après l'introduction à la gestion de la mémoire (section 6 page 40).

5.1.10 Constructeur de classe

Les classes étant elles-mêmes des objets manipulables en Objective-C, elles bénéficient également d'un constructeur que l'on peut redéfinir. Il s'agit bien sûr d'une méthode de classe (héritée de NSObject), de prototype `+(void) initialize ;`

Cette méthode est automatiquement appelée lors de la première utilisation de la classe, ou d'une de ses sous-classes. Attention cependant : il n'est pas tout à fait vrai de dire qu'elle n'est appelée qu'une fois pour une classe donnée ; en effet, si une classe dérivée ne redéfinit pas `+(void) initialize`, le mécanisme d'Objective-C rappelle le `+(void) initialize` de la classe mère.

5.2 Destructeurs

En C++, le destructeur, tout comme le constructeur, est une méthode particulière que l'on peut redéfinir. En Objective-C, c'est une méthode d'instance appelée `dealloc`.

En C++, le destructeur est appelé automatiquement quand on demande la libération d'un objet ; en Objective-C, c'est la même chose. Seule la façon de libérer l'objet change (cf. section 6 page 40).

Le destructeur ne devrait jamais être appelé explicitement. En réalité, il existe en C++ un seul cas où un destructeur peut être appelé explicitement : cela survient si le développeur gère lui-même le bassin de mémoire utilisé pour l'allocation. Mais en Objective-C, aucun cas ne justifie un appel explicite à `dealloc`. On peut utiliser des zones mémoires personnalisées grâce à Cocoa, mais leur utilisation ne perturbe pas les pratiques d'allocation/désallocation usuelles (cf. section 5.3 page suivante).

C++

```
class Point2D {
  public:
    ~Point2D();
};
Point2D::~Point2D() {}
```

Objective-C

```
@interface Point2D : NSObject
-(void) dealloc; //on peut redéfinir cette méthode
@end

@implementation Point2D
//dans cet exemple, ce n'était pas la peine de redéfinir dealloc
-(void) dealloc
{
  [super dealloc]; //penser à la super classe
}
@end
```

5.3 Opérateurs de copie

5.3.1 Clonage classique, `copy`, `copyWithZone:`, `NSCopyObject()`

En C++, il est important de définir une implémentation cohérente du constructeur par recopie et de l'opérateur d'affectation. En Objective-C, la surcharge d'opérateur étant impossible, on se contente du concept de clonage d'objet.

Le clonage est associé en Cocoa à un protocole (cf. section 4.4 page 23), du nom de `NSCopying` demandant l'implémentation de la méthode

```
-(id) copyWithZone:(NSZone*)zone;
```

dont l'argument, inattendu à première vue, est une zone mémoire dans laquelle allouer le clone. Cocoa permet en effet de gérer des réserves de mémoire personnalisées : les méthodes concernées peuvent prendre une zone mémoire en argument. Dans la plupart des cas, la zone mémoire par défaut est idéale, et il serait lourd de perpétuellement devoir la spécifier en argument. Heureusement, `NSObject` fournit une méthode

```
-(id) copy;
```

qui encapsule un simple appel à `copyWithZone:`, avec en paramètre la zone par défaut. Mais c'est bien `copyWithZone:` qui est déclarée dans `NSCopying`.

Enfin, la fonction utilitaire `NSCopyObject(...)` propose une approche un peu différente, qui peut alléger l'ensemble, mais nécessite aussi quelque prudence. Dans un premier temps, le code est présenté sans tenir compte de `NSCopyObject(...)` qui est expliquée en section 5.3.2 page suivante.

L'implémentation de `copyWithZone:` pour une classe quelconque Toto aura sans doute la forme suivante :

```
//si la superclasse n'implémente pas copyWithZone:, et que NSCopyObject()
//n'est pas utilisée
-(id) copyWithZone:(NSZone*)zone
{
  //il faut créer un nouvel objet
  Toto* clone = [[Toto allocWithZone:zone] init];
  //Les données d'instance doivent être recopiées "à la main"
  clone->entier = self->entier; //"entier" est ici de type "int"
  //déclencher le même mécanisme pour les sous-objets à cloner
  clone->objetACloner = [self->objetACloner copyWithZone:zone];
  //certains sous-objets n'ont pas à être copiés, mais partagés
  clone->objetAPartager = [self->objetAPartager retain]; //cf. gestion mémoire
  //s'il y a un accesseur en écriture, il peut être utilisé dans les deux cas
  [clone setObjet:self->objet];
  return clone;
}
```

Remarquez l'utilisation de `allocWithZone:` à la place d'un simple `alloc`, pour prendre en compte la zone passée en argument. En réalité, `alloc` encapsule un simple appel à `allocWithZone:` avec la zone par défaut. Pour en savoir plus sur la gestion de zones mémoires personnalisées, consultez plutôt une documentation Cocoa.

Il ne faut cependant pas oublier de tenir compte d'une implémentation éventuelle de `copyWithZone:` dans la superclasse.

```
//si la superclasse implémente copyWithZone:, et que NSCopyObject()
//n'est pas utilisée
-(id) copyWithZone:(NSZone*)zone
{
  Toto* clone = [super copyWithZone:zone]; //crée le nouvel objet
  //recopie des données d'instances spécifiques à la sous-classe courante
  clone->entier = self->entier; //"entier" est ici de type "int"
  //déclencher le même mécanisme pour les sous-objets à cloner
  clone->objetACloner = [self->objetACloner copyWithZone:zone];
  //certains sous-objets n'ont pas à être copiés, mais partagés
  clone->objetAPartager = [self->objetAPartager retain]; //cf. gestion mémoire
  //s'il y a un accesseur en écriture, il peut être utilisé dans les deux cas
  [clone setObjet:self->objet];
  return clone;
}
```

5.3.2 `NSCopyObject()`

La classe `NSObject` n'implémente pas le protocole `NSCopying`, c'est pourquoi le clonage d'une classe fille ne peut bénéficier d'un appel à `[super copy...]`, mais doit utiliser une initialisation plus classique par `[[... alloc] init]`.

La fonction utilitaire `NSCopyObject()` peut permettre d'écrire les choses plus simplement, mais implique de la prudence pour les données de type pointeur (objets inclus). Cette fonction crée la copie binaire d'un objet quelconque, et son prototype est le suivant :

```
//le nombre d'octets supplémentaires est habituellement 0, mais
//peut servir à prévoir l'espace pour les données d'instance de type tableau C.
id <NSObject> NSCopyObject(id <NSObject> unObjet,
                           unsigned int octetsSupplementaires, NSZone *zone)
```

La recopie binaire permet d'automatiser la recopie des données d'instance non-pointeurs ; en revanche, pour une donnée d'instance de type pointeur (objets inclus), il faut être conscient que cela crée une référence non comptabilisée vers la zone pointée. Il est donc d'usage de réinitialiser les pointeurs avec des valeurs provenant de clonages plus adaptés.

Notez que les *propriétés* (cf. section 12.2 page 61) ne sont pas automatiquement prises en compte par `NSCopyObject()` : c'est au développeur de fournir un clonage cohérent de ce qu'elles recouvrent.

```
//si la superclasse n'implémente pas copyWithZone:
-(id) copyWithZone:(NSZone*)zone
{
  Toto* clone = NSCopyObject(self, 0, zone);//recopie binaire des données
  //clone->entier = self->entier; //inutile : donnée binaire déjà recopiée

  //un sous-objet à cloner doit être réellement dupliqué
  clone->objetACloner = [self->objetACloner copyWithZone:zone];
  //un sous-objet partagé doit simplement être conscient de la nouvelle référence
  [clone->objetAPartager retain]; //cf. gestion mémoire

  //L'accesseur en écriture risque de libérer clone->objet. Cela serait ici
  //indésirable, du fait de la recopie binaire de la valeur du pointeur. Avant
  //d'utiliser l'accesseur, on procède donc à une réinitialisation du pointeur
  clone->objet = nil;
  [clone setObjet:self->objet];
  return clone;
}
```

```
//si la superclasse implémente copyWithZone:
-(id) copyWithZone:(NSZone*)zone
{
  Toto* clone = [super copyWithZone:zone];
  //est-ce qu'une des superclasses utilise NSCopyObject() ? De cela dépend
  //ce qu'il reste à faire !

  clone->entier = self->entier;//uniquement si NSCopyObject() n'a pas été utilisé
  //Dans le doute, on peut le faire systématiquement

  //un sous-objet à cloner doit être réellement dupliqué, cela ne change pas
  clone->objetACloner = [self->objetACloner copyWithZone:zone];

  //NSCopyObject() ou pas, un retain doit être fait (cf. gestion mémoire)
  clone->objetAPartager = [self->objetAPartager retain];

  clone->objet = nil; //dans le doute, autant procéder à cette réinitialisation
  [clone setObjet:self->objet];

  return clone;
}
```

5.3.3 Pseudo-clonage, mutabilité, `mutableCopy` et `mutableCopyWithZone`:

Si l'on clone un objet qui ne peut changer, une optimisation fondamentale consiste à ne pas réellement le dupliquer, mais à renvoyer une simple référence vers lui. Partant de ce principe, on distingue la notion d'objet *non modifiable* et d'objet *modifiable* (*mutable*).

Un objet non modifiable n'a aucune méthode permettant de changer la valeur de ses données d'instance ; seul le constructeur lui a donné un état. Dans ce cas, on peut sans risques le « pseudo-cloner » en ne renvoyant comme clone qu'une simple référence vers lui-même. Comme ni l'original ni son clone ne peuvent être modifiés, aucun des deux ne risque d'être silencieusement affecté par une perturbation de l'autre. Une implémentation très efficace de `copyWithZone:` peut donc être proposée pour ce cas précis.

```
-(id) copyWithZone:(NSZone*)zone
{
  //renvoie l'objet lui-même (référencé une fois de plus)
  return [self retain]; //voyez la section sur la gestion mémoire
}
```

L'utilisation de `retain` découle de la gestion de la mémoire en Objective-C (cf. section 6 page 40). On incrémente de 1 le compteur de références de l'objet pour officialiser l'existence du clone, de telle sorte qu'une demande de suppression de ce dernier n'entraîne pas la suppression de l'original.

Ce « pseudo-clonage » n'est pas une optimisation marginale. La création d'un objet demande une allocation de mémoire, ce qui est un processus « long » qu'il vaut mieux éviter si c'est possible. Il est donc intéressant de classer les objets en deux familles : les objets non modifiables, pour lesquels on est sûr que le clonage peut être fictif, et les autres. Pour réaliser la distinction, il suffit de créer des classes « non-mutables », et d'éventuellement les dériver en versions « mutables », par adjonction de méthodes permettant de modifier les données d'instance. Par exemple, en Cocoa, `NSMutableString` dérive de `NSString`, `NSMutableArray` dérive de `NSArray`, `NSMutableData` dérive de `NSData`, etc.

Cependant, avec les techniques présentées jusqu'ici, il semble impossible d'obtenir un véritable clone, modifiable, d'un objet non modifiable qui ne saurait que se « pseudo-cloner ». Une telle

lacune limiterait grandement l'intérêt des objets non modifiables en les isolant trop du monde extérieur.

En plus du protocole `NSCopying`, il existe donc un protocole (cf. section 4.4 page 23) du nom de `NSMutableCopying`, demandant l'implémentation de

```
-(id) mutableCopyWithZone:(NSZone*)zone;
```

La méthode `mutableCopyWithZone`: doit renvoyer un clone modifiable, dont les perturbations n'affectent pas l'original. De façon similaire à la méthode `copy`, il existe une méthode `mutableCopy` qui simplifie l'appel à `mutableCopyWithZone`: en lui passant automatiquement la zone par défaut en argument. L'implémentation de `mutableCopyWithZone`: aura sans doute la forme suivante, similaire à celle présentée pour la copie classique dans la section précédente :

```
//si la superclasse n'implémente pas mutableCopyWithZone:
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Toto* clone = [[Toto allocWithZone:zone] init];//ou NSCopyObject() si possible
  clone->entier = self->entier;
  //Tout comme avec copyWithZone:, les sous-objets peuvent être
  //clonés ou partagés selon le cas. Un sous-objet mutable peut être
  //cloné avec un appel en cascade à mutableCopyWithZone:
  //...
  return clone;
}
```

Ne toujours pas oublier de tenir compte des implémentations éventuelles de `mutableCopyWithZone`: dans une superclasse :

```
//si la superclasse implémente mutableCopyWithZone:
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Toto* clone = [super mutableCopyWithZone:zone];
  //...
  return clone;
}
```

6 Gestion mémoire

6.1 `new` et `delete`

Les mots-clefs `new` et `delete` du C++ n'existent pas en Objective-C (`new` existe sous forme de méthode, mais il ne s'agit que d'un raccourci déconseillé pour `alloc`+`init`). Ils sont respectivements remplacés par un appel à `alloc` (cf. section 5.1 page 29) et à `release` (cf. section 6.2 de la présente page).

6.2 Compteur de références

La gestion mémoire en Objective-C est un des points les plus importants du langage, et un de ses points forts. En langage C ou C++, une zone mémoire est allouée une fois et détruite une fois. On peut y faire de multiples références en utilisant le nombre de pointeurs approprié. Sur un seul des pointeurs sera effectué `delete`.

L'Objective-C implémente quant à lui un système de comptage de références. L'objet est conscient du nombre de liens dirigés vers lui. On peut expliquer ce principe par la métaphore du chien et des laisses (exemple directement tiré de *Cocoa Programming for MacOS X* [7]). Si l'objet est un chien, chacun peut réclamer une laisse pour le maintenir en place. Si quelqu'un se désintéresse du chien, il peut lâcher la laisse. Tant que le chien a au moins une laisse, il est contraint de rester en place. Mais dès que le nombre de laisses tombe à 0, le chien est... libéré !

Plus techniquement, le compteur de référence d'un objet nouvellement créé est fixé à 1. Si une portion du code nécessite de référencer cet objet durablement, elle lui envoie un message `retain`, qui incrémente de 1 le compteur. Si une portion de code référençant l'objet n'en a plus l'utilité, elle lui envoie un message `release`, qui décrémente le compteur de 1.

Un objet peut recevoir un nombre quelconque de `retain` et de `release`, tant que le compteur de références a une valeur strictement positive. En effet dès qu'il tombe à 0, le destructeur `dealloc` est automatiquement appelé. Envoyer de nouveaux `release` à l'adresse, devenue invalide, de l'objet, provoque une faute mémoire.

De cette manière, le développeur n'a plus à se soucier de centraliser la destruction auprès d'un « propriétaire privilégié ». Des objets qui en référencent un même autre, c'est le dernier à relâcher le lien qui provoque sa destruction. Partager une référence à un objet devient beaucoup plus aisé, moins propice aux fuites mémoires ou aux destruction multiples (et donc invalides).

Notez que cette technique n'est pas équivalente aux `auto_ptr` de la STL du C++. En revanche, la bibliothèque Boost [5] propose une encapsulation des pointeurs nommée `shared_ptr`, qui implémente le comptage de références. Elle ne fait cependant pas partie du standard et reste lourde à manipuler.

6.3 `alloc`, `copy`, `mutableCopy`, `retain`, `release`

Savoir comment fonctionne le gestionnaire mémoire n'explique pas complètement comment il s'utilise. Le but de cette section est de donner quelques règles fondamentales d'utilisation. Le mot-clef `autorelease` est mis à part à dessein, car il est assez subtil à comprendre.

La règle de base à appliquer est **Tout responsable de l'incrémentation du compteur de référence, *via* `alloc`, `[mutable]copy[WithZone :]` ou `retain` est chargé d'appliquer le ou les `[auto]release` correspondants**. Ce sont effectivement les trois façons d'incrémenter le compteur de références. Cela signifie qu'il ne faut se préoccuper de faire un `release` que :

- si on instancie explicitement un objet avec `alloc` ;
- si on crée explicitement une copie de cet objet avec `copy[WithZone :]` ou `mutableCopy[WithZone :]` (que la copie soit techniquement une duplication ou, si elle est non mutable, que ce soit un simple pointeur vers l'objet initial, n'a aucune importance, cela doit rester transparent ; cf. section 5.3.3 page 38) ;
- si on effectue un `retain` explicite.

Rappelez-vous que par défaut, il est légal d'envoyer un message (comme `release`) à `nil`, sans aucune conséquence (voyez la section 3.4.1 page 20).

6.4 `autorelease`

6.4.1 Indispensable `autorelease`

La règle citée dans la section précédente est si importante qu'il n'est pas inutile de la répéter : **Tout responsable de l'incrémentation du compteur de référence, *via* `alloc`, `[mutable]copy[WithZone :]` ou `retain` est chargé d'appliquer le ou les `[auto]release` correspondants**.

Assurément, avec les seuls `alloc`, `retain` et `release`, cette règle serait absolument inapplicable. En effet, il existe des fonctions qui ne sont pas des constructeurs, mais qui ont pour but de construire des objets : par exemple un opérateur binaire d'addition en C++(`obj3 operator+(obj1, obj2)`). Dans le cas du C++, l'objet renvoyé par la fonction est déposé sur la pile, et sera détruit automatiquement lorsqu'il deviendra caduc. Or, en Objective-C, les objets automatiques n'existent pas. Une telle fonction a donc forcément utilisé un `alloc`, mais ne peut détruire l'objet avant de le déposer sur la pile ! Ci-après sont illustrés quelques exemples problématiques d'utilisation.

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  return result;
}
//ERREUR : la fonction effectue un "alloc", donc elle produit
//un objet dont le compteur de référence vaut 1. Pourtant,
//d'après la règle, elle est chargée de le supprimer.

//Exemple de fuite mémoire induite en cas d'addition de trois Point2D:
[calculator add:[calculator add:p1 and:p2] and:p3];

//le résultat de la première addition est anonyme, donc personne
//ne peut le libérer. C'est une fuite mémoire.
```

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  return [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                               andY:([p1 getY] + [p2 getY])];
}
//ERREUR : c'est exactement le même code que précédemment.
//Ne pas utiliser de variable intermédiaire ne change absolument
//rien au problème.
```

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result release];
  return result;
}
//ERREUR : bien sûr, il ne rime à rien de créer l'objet puis de le
//détruire.
```

Le problème semble insoluble ; il le serait si `autorelease` n'existait pas. Pour faire simple, disons qu'envoyer `autorelease` à un objet revient à lui envoyer un `release` qui sera appliqué « un peu plus tard ». Attention, « un peu plus tard » ne veut pas dire « n'importe quand » ; le détail est

donné en section 6.4.2. Dans un premier temps, montrons son utilisation dans la seule solution possible :

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result autorelease];
  return result; //une écriture plus courte serait "return [result autorelease]"
}
//CORRECT : "result" sera automatiquement libéré plus tard,
//après son éventuelle utilisation dans toute fonction appelante.
```

6.4.2 Bassin d'autorelease

D'après la section précédente, le `autorelease` est en quelque sorte un `release` magique qui sera appliqué au bon moment. Il ne serait cependant ni raisonnable ni efficace de laisser deviner au compilateur quel est ce bon moment. Autant utiliser un ramasse-miettes dans ce cas. Pour y remédier, il convient d'expliquer comment fonctionne `autorelease`.

A chaque fois qu'un objet reçoit un `autorelease`, il est inscrit dans un bassin de désallocation automatique (*autorelease pool*). Quand ce bassin est vidé, l'objet reçoit un `release` effectif. Le problème s'est déplacé : comment gère-t-on ce bassin ?

La réponse est double : si vous utilisez Cocoa pour une application avec interface graphique, la plupart du temps, vous n'avez rien besoin de faire. Sinon, il vous faut créer le bassin et le vider vous même.

Une application avec interface utilisateur utilise une *boucle événementielle*. C'est une boucle qui attend une action de l'utilisateur pour réveiller le programme et lui transmettre cette action, avant de se remettre en sommeil jusqu'au prochain événement. En Cocoa, lorsque l'on programme une telle application, un bassin d'autorelease est automatiquement créé à chaque début de passage dans la boucle, et vidé à la fin. C'est tout à fait logique : généralement, une action de l'utilisateur provoque une succession de tâches. Des objets temporaires sont créés, puis détruits, car ils n'ont pas besoin d'être conservés pour la suite des événements. Si certains d'entre eux doivent l'être, c'est au développeur de prévoir les `retain` nécessaires.

En revanche, s'il n'y a pas d'interface utilisateur, il faut créer un bassin d'autorelease en amont du code le nécessitant. Lorsqu'un objet reçoit `autorelease`, il sait automatiquement trouver le bassin d'allocation actuellement déclaré, il n'y a pas besoin de le lui spécifier. Ensuite, quand il est opportun de vider le bassin, il faut en fait le détruire, avec un simple `release` : sa destruction entraîne en cascade les `release` sur chaque objet qu'il contient. Typiquement, un programme Cocoa destiné à la ligne de commande contient :

```
int main(int argc, char* argv[])
{
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
  //...
  [pool release];
  return 0;
}
```

Notez que MacOS 10.5 apporte la méthode `drain` à la classe `NSAutoreleasePool`. Cette méthode est synonyme de `release` si le ramasse-miettes est désactivé, et effectue un appel au ramasse-miettes dans le cas contraire (cf. section 6.6 page 50). Cela permet d'uniformiser un code qui devrait se comporter de la même façon dans les deux cas.

6.4.3 Utilisation de plusieurs bassins d'autorelease

Il est possible, et parfois utile, d'avoir plusieurs bassins d'autorelease dans le même programme. Un objet recevant `autorelease` s'inscrira dans le dernier bassin créé. De ce fait, si une fonction crée et utilise un grand nombre d'objets temporaires, un gain de performance peut être obtenu en créant un bassin d'autorelease local. De cette manière, la foule d'objets temporaires sera détruite au plus vite, et n'encombrera pas la mémoire inutilement.

6.4.4 Prudence avec `autorelease`

Ce n'est pas parce qu'`autorelease` est pratique qu'il doit être mal exploité.

- D'abord, envoyer plus de `autorelease` que nécessaire est aussi néfaste que d'envoyer trop de `release` : cela provoque une faute mémoire lors du vidage du bassin ;
- Ensuite, même s'il est vrai que dans un programme Objective-C, n'importe quel message `release` peut être remplacé par un message `autorelease`, cela serait au prix d'une baisse de performance, car le bassin d'autorelease rajoute un intermédiaire à qu'un `release` normal. De plus, remettre à plus tard toute libération tendrait à provoquer des pics d'utilisation mémoire inutiles et malvenus.

6.4.5 `autorelease` et `retain`

Grâce à `autorelease`, une méthode qui crée un objet est capable de planifier elle-même la libération. Pourtant, il est courant que l'objet construit doive être conservé. Dans ce cas, il suffit d'appliquer un `retain` sur cet objet renvoyé par la fonction ; mais il faut alors planifier un `release` au moment opportun. Il y a donc deux points de vue autour de la vie de cet objet :

- du point de vue de l'implementation de la fonction, l'objet est créé et sa destruction planifiée ;
- du point de vue de l'utilisateur de la fonction, on prolonge la durée de vie de l'objet grâce au `retain` (l'appel à `autorelease` planifié par la fonction ne fera plus tomber le compteur de références à 0), mais puisqu'on a incrémenté le compteur de 1, on devient responsable de la décrémentation correspondante.

6.4.6 Constructeurs de commodité, constructeurs virtuels

Le principe de l'application successive de `alloc` puis de `init` lors de l'instanciation d'une classe est laborieux dans certains cas. Il peut fort heureusement être court-circuité par ce qu'on appelle un *constructeur de commodité*. Un tel constructeur, dont le nom devrait toujours être préfixé par celui de la classe, a le même rôle qu'une méthode d'`init`, mais il fait le `alloc` lui-même. Cependant, l'objet renvoyé est inscrit au bassin d'autorelease, et sera donc temporaire s'il ne lui est pas envoyé un `retain`. Exemple d'utilisation :

```
//laborieux
NSNumber* zero_a = [[NSNumber alloc] initWithFloat:0.0f];
...
[zero_a release];

...

//plus sympathique
NSNumber* zero_b = [NSNumber numberWithFloat:0.0f];
...
//pas besoin de release
```

Si vous avez lu la section consacrée à la gestion de la mémoire (section 6 page 40), vous devinez qu'un tel constructeur repose sur `autorelease`. Le code correspondant n'est cependant pas évident, car il nécessite d'utiliser `self` correctement. En effet, un constructeur de commodité est une *méthode de classe*, donc `self` se réfère à un objet `Class`, qui est une instance de *méta-classe*. Dans un

initialisateur, qui est une méthode *d'instance*, `self` est une *instance* de la classe, et donc se réfère à un objet « normal ».

Il est facile d'écrire un mauvais constructeur de commodité. Supposons pour l'exemple l'existence d'une classe `Vehicule` dotée d'une couleur et d'un constructeur de commodité.

```
//La classe Vehicule
@interface Vehicule : NSObject
{
  NSColor* couleur;
}
-(void) setCouleur:(NSColor*)couleur;

//constructeur de commodité
+(id) vehiculeDeCouleur:(NSColor*)couleur;
@end
```

L'implémentation du constructeur de commodité est assez particulière.

```
//Mauvais constructeur de commodité
+(Vehicule*) vehiculeDeCouleur:(NSColor*)couleur
{
  // la valeur de "self" ne devrait pas changer ici !
  self = [[self alloc] init]; // ERREUR !
  [self setCouleur:couleur];
  return [self autorelease];
}
```

`self` dans cette méthode de classe se réfère à la *classe*. Il ne faut donc pas lui attribuer une *instance*.

```
//Constructeur presque parfait
+(id) vehiculeDeCouleur:(NSColor*)couleur
{
  id nouvelleInstance = [[Vehicule alloc] init]; // OK, mais les éventuelles
                                                 // sous-classes sont ignorées
  [nouvelleInstance setCouleur:couleur];
  return [nouvelleInstance autorelease];
}
```

Une dernière subtilité subsiste : il est possible en Objective-C d'avoir un constructeur virtuel. Il suffit que le constructeur de commodité fasse une introspection pour savoir quel est le type de classe réel de l'objet exécutant la méthode. Dans ce cas, il produit directement un objet du bon type dérivé. On a besoin pour cela du faux mot-clef `class`, qui est en fait une méthode de `NSObject` renvoyant l'objet classe de l'objet courant (l'instance de la méta-classe).

```
@implementation Vehicule
+(id) vehiculeDeCouleur:(NSColor*)couleur
{
  id nouvelleInstance = [[[self class] alloc] init]; // PARFAIT, la classe est
                                                     // identifiée dynamiquement
  [nouvelleInstance setCouleur:couleur];
  return [nouvelleInstance autorelease];
}
@end

@interface Voiture : Vehicule {...}
@end

...

//produit une voiture (rouge) !
id voiture = [Voiture vehiculeDeCouleur:[NSColor redColor]];
```

Tout comme pour la règle du préfixe **init** sur les initialisateurs, il est très déconseillé de ne pas préfixer un constructeur de commodité par le nom de la classe. Il n'y a que peu de situations pour lesquelles cette règle est contournée, comme par exemple le `[NSColor redColor]` du code précédent, qui aurait dû s'écrire `[NSColor colorRed]`.

Enfin, rappelons la loi : **Tout responsable de l'incrémentation du compteur de référence, *via* `alloc`, `[mutable]copy[withZone :]` ou `retain` est chargé d'appliquer le ou les `[auto]release` correspondants**. En **appelant** un constructeur de commodité, on n'effectue pas soi-même le `alloc`, donc on n'est pas chargé du `release`. Par contre, lorsqu'on **écrit** un tel constructeur, on écrit `alloc`, donc on doit bien mettre `autorelease`.

6.4.7 Accesseurs en écriture (mutateurs)

Un accesseur en écriture, ou *mutateur*, est l'exemple typique de ce que l'on ne sait pas écrire quand on ne connaît pas la gestion de la mémoire en Objective-C. Supposons une classe encapsulant une `NSString` appelée *chaine*, et supposons que l'on veuille changer la valeur de cette chaîne de caractères. Cet exemple anodin suffit à soulever le problème principal de la création des mutateurs : comment doit-on traiter l'objet passé en paramètre ? Contrairement au C++, un seul prototype est envisageable (un objet ne peut être passé que par pointeur), mais plusieurs implémentations sont possibles. Il peut s'agir d'une *assignation*, d'une *assignation avec retenue*, ou d'une *recopie*. Chaque implémentation à une signification particulière vis-à-vis du modèle de données choisi par le développeur. De plus, dans ces trois cas, pour éviter une fuite mémoire, il faut libérer les anciennes ressources avant de les écraser.

assignation (code réduit)
On ne retient qu'une référence vers l'objet extérieur, sans lien fort comme un `retain`. Si l'objet extérieur est modifié, cela est visible dans la classe courante. Si l'objet extérieur venait à être détruit sans que la présente référence soit mise à `nil` au préalable, elle deviendrait invalide.

```
-(void) setString:(NSString*)nouvelleChaine
{
   ... gestion mémoire à préciser
  self->chaine = nouvelleChaine; //assignation

}
```

assignation avec retenue (code réduit)
On ne retient qu'une référence vers l'objet extérieur, et le compteur de référence est incrémenté par un `retain`. Si l'objet extérieur est modifié, cela est visible dans la classe courante. L'objet extérieur ne peut pas être détruit tant que subsiste au moins la référence courante.

```
-(void) setString:(NSString*)nouvelleChaine
{
   ... gestion mémoire à préciser
  self->chaine = [nouvelleChaine retain]; //assignation avec retenue
}
```

copie (code réduit)
On ne retient pas une référence vers l'objet extérieur, mais vers un clone créé pour l'occasion. Si l'objet extérieur est modifié, cela n'est pas visible sur le clone. Logiquement, le clone est géré par l'objet courant qui le crée et le contient, et ne devrait pas lui survivre.

```
-(void) setString:(NSString*)nouvelleChaine
{
   ... gestion mémoire à préciser
  self->chaine = [nouvelleChaine copy]; //clonage;
                                        //le protocole NSCopying est employé
}
```

Pour compléter le code, il faut tenir compte de l'état antérieur de l'objet : dans chaque cas, le mutateur doit au préalable libérer l'éventuelle ancienne référence avant d'en attribuer une nouvelle. Cette partie du code est propice aux erreurs car de nombreux pièges doivent être évités.

assignation (code complet)
C'est le cas le plus simple. L'ancienne référence peut être écrasée.

```
-(void) setString:(NSString*)nouvelleChaine
{
  //aucune lien fort : l'ancienne référence peut être écrasée.
  self->chaine = nouvelleChaine; //assignation
}
```

assignation avec retenue (code complet)
Dans cette situation, il faut penser à libérer l'ancienne référence, sauf en cas d'égalité avec la nouvelle.

```
//Codes incorrects

-(void) setString:(NSString*)nouvelleChaine
{
  self->chaine = [nouvelleChaine retain];
  //ERREUR : fuite mémoire : l'ancien "chaine" n'est plus référencé
}

-(void) setString:(NSString*)nouvelleChaine
{
  [self->chaine release];
  self->chaine = [nouvelleChaine retain];
  //ERREUR : si nouvelleChaine == chaine,  (cela peut arriver),
  //et que le compteur de références de nouvelleChaine était à 1,
  //alors il est invalide d'utiliser nouvelleChaine (chaine) après
  //[self->chaine release], parce qu'elle a été désallouée à cet instant
}

-(void) setString:(NSString*)nouvelleChaine
{
  if (self->chaine != nouvelleChaine)
    [self->chaine release]; //ok: il est autorisé d'envoyer release même à nil
  self->chaine = [nouvelleChaine retain];  //ERREUR : devrait être dans le "if";
  //car dans le cas où chaine == nouvelleChaine,
  //il ne faut pas incrémenter le compteur
}
```

```
//Codes corrects

//Méthode "Vérifier avant de modifier"
//la méthode la plus intuitive pour un développeur C++
-(void) setString:(NSString*)nouvelleChaine
{
  //éviter le cas dégénéré où il n'y a rien à faire
  if (self->chaine != nouvelleChaine)
  {
    [self->chaine release]; //libérer l'ancien
    self->chaine = [nouvelleChaine retain]; //retenir le nouveau
  }
}

//Méthode "Autorelease de l'ancienne valeur"
-(void) setString:(NSString*)nouvelleChaine
{
  [self->chaine autorelease]; //même si chaine == nouvelleChaine,
                        //c'est correct, puisque le release est retardé...
  self->chaine = [nouvelleChaine retain];
   //... et que ce retain intervient donc avant
}

//Méthode "retenir avant de libérer"
-(void) setString:(NSString*)nouvelleChaine
{
  [self->nouvelleChaine retain]; //on incrémente le compteur de 1 (sauf sur nil)
  [self->chaine release]; //...pour qu'il ne risque pas de tomber à zero ici
  self->chaine = nouvelleChaine; //mais on ne réexécute pas de "retain" ici !
}
```

copie (code complet)
Tant pour les erreurs courantes que les bonnes solutions, cette situation est quasiment identique au cas de l'assignation avec retenue, où le `retain` est remplacé par `copy`.

pseudo-clonage
Notez aussi que l'opération de copie peut en réalité correspondre à un « pseudo-clonage » (cf. section 5.3.3 page 38), sans que cela ait la moindre incidence.

6.4.8 Accesseurs en lecture

Les objets étant toujours alloués dynamiquement en Objective-C, ils sont référencés et encapsulés sous forme de pointeurs. Typiquement, les accesseurs en lecture des objets Objective-C ne font que renvoyer la valeur du pointeur, et ne recopient pas l'objet à la volée. Notez que le nom d'un accesseur est usuellement celui de la donnée concernée, cela est possible en Objective-C et ne provoque pas de conflit. Dans le cas d'un booléen, le nom peut être précédé de `is` comme pour répondre à un prédicat.

```
@interface Button
{
  NSString* label;
  BOOL      pressed;
}
-(NSString*) label;
-(void) setLabel:(NSString*)newLabel;
-(BOOL) isPressed;
@end

@implementation Button
-(NSString*) label
{
  return label;
}

-(BOOL) isPressed
{
  return pressed;
}

-(void) setLabel:(NSString*)newLabel {...}
@end
```

Lorsque vous retournez des pointeurs de données d'instance, il devient aisé de modifier ces données si elles sont de classe mutable. Il peut être indésirable de permettre de telles modifications depuis l'extérieur, la protection des données étant un point important.

```
@interface Button
{
  NSMutableString* label;
}
-(NSString*) label;
@end

@implementation Button
-(NSString*) label
{
  return label; //OK, mais l'utilisateur bien informé peut downcaster
  //le résultat en NSMutableString, et ainsi modifier la chaine
}

-(NSString*) label
{
  //possibilité 1 :
  return [NSString stringWithString:label];
  //OK : une nouvelle chaine, non mutable, est renvoyée

  //possibilité 2 :
  return [[label copy] autorelease];
  //OK, utiliser copy (et non mutableCopy) sur une NSMutableString renvoie
  //bien une NSString
}
@end
```

6.5 Cycles de `retain`

Un cycle de `retain` est une situation à éviter. Si par exemple un objet A « retient » (*retain*) un objet B, et que B et C se retiennent l'un-l'autre, alors B and C forment un cycle. Un cycle peut contenir plus de deux objets, cet exemple est juste le plus simple.

$$A \rightarrow B \rightleftarrows C$$

Si A « relâche » (*release) B*, B ne sera pas désalloué, puisqu'il est retenu par C. Et C ne sera pas désalloué non plus puisqu'il est retenu par B. Or, A était la seule référence vers ce cycle, il n'est donc plus accessible. Les cycles de `retain` engendrent donc habituellement des fuites mémoire.

C'est la raison pour laquelle, dans une structure d'arbre par exemple, un nœud « retient » (*retain*) ses descendants, mais que ceux-là ne devraient pas faire de même avec leur parent.

6.6 Ramasse-miettes

Objective-C 2.0 (cf. section 1.3 page 6) implémente un ramasse-miettes (*garbage collector*). Autrement dit, il est possible de déléguer toute la gestion mémoire et ne plus se soucier des *retain/release*. Une des forces d'Objective-C 2.0 est d'avoir fait du ramasse-miettes une fonctionnalité **optionnelle** : on peut décider de ne pas l'activer, pour contrôler finement le cycle de vie des objets, ou de l'activer et d'écrire un code moins susceptible d'être buggué. Le ramasse-miettes est activé ou non pour l'ensemble du programme.

Si le ramasse-miettes est activé, `retain`, `release` et `autorelease` sont redéfinis pour ne rien faire. Ainsi, un code écrit sans tenir compte du ramasse-miettes peut en théorie être recompilé sans modifications, la gestion mémoire étant alors entièrement déléguée. « En théorie », car quelques subtilités doivent être prises en compte sur la libération des ressources. Il est donc non trivial, voire déconseillé, d'écrire un code parfaitement adapté aux deux cas.

Ces subtilités sont décrites dans un document d'Apple [2]. Ci-dessous ne sont cités que quelques particularités mettant en évidence des points importants, pour lesquels la précédente référence doit être consultée.

6.6.1 `finalize`

Dans un environnement utilisant le ramasse-miettes, le non-déterminisme de l'ordre de destruction des objets rend inadaptée la simple utilisation de `dealloc`. Une méthode `finalize` a été ajoutée à `NSObject` pour scinder la destruction des objets en plusieurs étapes : la libération des ressources et la destruction effective. Mais une « bonne » méthode `finalize` doit respecter quelques constraintes de conception. Celles-ci sont détaillées en [2].

6.6.2 `weak`, `strong`

Il est rare de rencontrer *__weak* et *__strong* utilisés explicitement dans une déclaration. La connaissance de leur utilité permet cependant de comprendre une nouvelle difficulté liée au ramasse-miettes.

Par défaut, un pointeur d'objet utilise l'attribut *__strong* : c'est une référence **forte**. Cela signifie que l'objet pointé ne peut être détruit tant que subsiste au moins cette référence. C'est le comportement attendu : quand toutes les références (fortes) ont disparu, l'objet peut être collecté par le ramasse-miettes. Dans certains cas, il faut pouvoir désactiver ce comportement : certaines collections ne devraient pas prolonger la durée de vie des objets qu'elles contiennent, au risque d'empêcher que ces objets soient jamais détruits. Dans ce cas, l'implémentation de ces collections utilise des références **faibles**, c'est-à-dire utilisant l'attribut *__weak*. `NSHashTable` en est un exemple (cf. section 11.1 page 57). Une référence *__weak* passe automatiquement à `nil` quand l'objet pointé a disparu.

Un exemple très pertinent de l'utilisation des références faibles est le *Notification Center* de Cocoa, qui sort donc du cadre du seul Objective-C, et n'est pas détaillé ici.

6.6.3 NSMakeCollectable()

`Cocoa` n'est pas la seule API de MacOS X. `Core Foundation` en est une autre ; elles sont compatibles, peuvent échanger très facilement des données et objets, mais `Core Foudation` est une API procédurale écrite en C uniquement. *A priori*, le ramasse-miettes ne peut fonctionner avec les pointeurs de `Core Foundation`. Ce problème a cependant été pris en compte, et il est possible d'utiliser `Core Foundation` sans fuites mémoires dans un environnement utilisant le ramasse-miettes. La documentation de `NSMakeCollectable` est un bon point d'entrée pour cerner le problème.

6.6.4 AutoZone

Le ramasse-miettes créé pour Objective-C par Apple se nomme `AutoZone`. Il a été rendu disponible en Open-Source [1]. Des évolutions sont prévues avec MacOS X10.6.

7 Exceptions

La gestion des exceptions en Objective-C est plus proche de celle du Java que de celle du C++ par l'adjonction du `@finally`. Le `finally` est connu en Java mais pas en C++. Il s'agit d'une clause supplémentaire (mais optionnelle) à un bloc `try()...catch()`, contenant du code qui sera exécuté dans tous les cas, qu'une exception soit interceptée ou non. Cela est généralement très utile pour une libération concise et propre des ressources.

Le comportement du `@try...@catch...@finally` en Objective-C est par ailleurs très classique ; en revanche, on ne peut « lancer » (*throw*) que des objets (contrairement au C++). Un exemple d'utilisation avec et sans `@finally` est présenté ci-après.

Sans finally	Avec finally

`BOOL probleme = YES;`
`@try{`
`  actionRisquee();`
`  probleme = NO;`
`}`
`@catch (MonException* e){`
`  faireUnTruc();`
`  nettoyage();`
`}`
`@catch (NSException* e){`
`  faireUnAutreTruc();`
`  nettoyage();`
`  //pour l'exemple, on relance l'exception`
`  @throw`
`}`
`if (!probleme)`
`  nettoyage();`
	`@try{`
	`  actionRisquee();`
	`}`
	`@catch (MonException* e){`
	`  faireUnTruc();`
	`}`
	`@catch (NSException* e){`
	`  faireUnAutreTruc();`
	`  @throw //relancer l'exception`
	`}`
	`@finally{`
	`  nettoyage();`
	`}`

On peut se passer du `@finally`, mais il est un bon atout pour une gestion propre des exceptions. Comme le montre l'exemple ci-dessus, il gère même le cas où une exception est relancée dans un `@catch`. En fait, le `@finally` est exécuté dès que l'on sort de la portée du `@try`. Ci-après s'en trouve une illustration.

```
int f(void)
{
  printf("f: 1-on me voit\n");
  //Voyez la section sur les chaînes de caractères pour comprendre
  //la syntaxe avec le "@"
  @throw [NSException exceptionWithName:@"kaput"
                                reason:@"c'est la faute à Rousseau"
                                userInfo:nil];
  printf("f: 2-on ne me voit pas\n");
}

int g(void)
{
  printf("g: 1-on me voit\n");
  @try {
    f();
    printf("g: 2-on ne me voit pas (dans cet exemple)\n");
  }
  @catch(NSException* e) {
    printf("g: 3-on me voit\n");
    @throw;
    printf("g: 4-on ne me voit jamais\n");
  }
  @finally {
    printf("g: 5-on me voit\n");
  }
  printf("g: 6-on ne me voit pas (dans cet exemple)\n");
}
```

Enfin, le `catch(...)` du C++ (avec des points de suspension), chargé de tout intercepter, n'existe pas en Objective-C. Il est en effet inutile, puisque comme seuls des objets peuvent être lancés, on peut toujours les intercepter avec le type `id` (cf. section 3.1 page 10).

Notez qu'une classe `NSException` existe en Cocoa, et qu'il est fortement conseillé d'en faire dériver tout objet que l'on désire lancer. Aussi, un `catch(NSException* e)` devrait être l'équivalent idéal d'un `catch(...)`.

8 Multithreading

8.1 Thread-safety

Rien n'empêche un programme Objective-C d'utiliser les API [2] POSIX pour réaliser du multithreading. Cocoa propose également ses propres classes pour gérer des fils d'exécutions parallèles. Les problèmes restent identiques : il faut garantir que certaines portions de code, exécutées simultanément, ne donnent pas des résultats imprévisibles en opérant sur le même espace mémoire.

Les APIs POSIX, tout comme Cocoa, fournissent pour cela les outils indispensables de verrous et mutex. Objective-C propose en sus un mot-clef, `@synchronized`, équivalent au mot-clef Java du même nom.

8.2 `@synchronized`

Une section de code encadrée d'un bloc `@synchronized(...)` est automatiquement verrouillée pour n'être accessible que par un thread à la fois. Ce n'est pas la meilleure solution dans tous les cas, mais elle simplifie et allège le code de la plupart des sections critiques.

`@synchonized` requiert un objet (quelconque, par exemple `self`) à utiliser comme verrou.

```
@implementation MaClass

-(void) methodeCritique:(id)unObjet
{
  @synchronized(self)
  {
    //code protégé de tous les autres blocs utilisant @synchronized(self)
    //(avec le même "self"...)
  }
  @synchronized(unObjet)
  {
    //code protégé de tous les autres blocs utilisant @synchronized(unObjet)
    //(avec le même "unObjet"...)
  }
}

@end
```

2. Application Programming Interface

9 Chaînes de caractères en Objective-C

9.1 Seuls objets statiques possibles d'Objective-C

En langage C, les chaînes de caractères sont des tableaux de `char`, ou des pointeurs `char*`. La gestion de ces chaînes est difficile et source de bien d'erreurs. La classe `string` de C++ est un véritable soulagement. En Objective-C, il a été expliqué dans la section 5 page 29 que les objets ne peuvent être automatiques, et doivent être alloués à l'exécution du programme. Cela est incompatible avec l'utilisation de chaînes statiques. On en serait réduit à avoir des chaînes C statiques passées en paramètre de construction d'objets `NSString`, ce qui ferait double emploi, ajoutant lourdeur et allocations mémoires indésirables.

Heureusement, il existe des chaînes Objective-C statiques. Ce sont de simples chaînes C entre guillemets, mais précédées de `@`.

```
NSString* pasPratique = [[NSString alloc] initWithUTF8String:"helloWorld"];
NSString* toujoursPasPratique = //initWithFormat est une sorte de sprintf()
   [[NSString alloc] initWithFormat:@"%s", "helloWorld"];
NSString* pratique = @"hello world";
```

En outre, on peut appeler les méthodes d'une chaîne statique comme sur tout objet.

```
int taille = [@"hello" length];
NSString* helloMajuscule = [@"hello" uppercaseString];
```

9.2 `NSString` et les encodages

Les chaînes Objective-C sont précieuses, car en plus d'implémenter un grand nombre de méthodes, elles permettent également de supporter de multiples encodages (ASCII, Unicode, ISO Latin 1...). Cela permet notamment une traduction (localisation) des applications grandement simplifiée.

9.3 Description d'un objet, extension de format `%@`, d'une NSString à une chaîne C

En Java, tout objet hérite de la classe `Object` et bénéficie d'une méthode `toString()`, destinée à décrire cet objet par une chaîne de caractères bien pratique pour le débogage. En Objective-C, cette méthode s'appelle `description` et renvoie une `NSString`.

Le `printf` du langage C n'a pas été étendu pour supporter les `NSString`. On lui préférera `NSLog` ou toute fonction acceptant une chaîne de format du style de `printf`. Pour une `NSString`, le format à passer n'est pas « %s » mais « %@ ». En fait, « %@ » peut être utilisé pour n'importe quel type d'objet, pusique cela affichera en réalité le résultat d'un appel à `-(NSString*) description`.

En outre, une `NSString` peut facilement être convertie en chaîne C par la méthode `UTF8String` (anciennement `cString`).

```
char* nom = "Perrine";
NSString* parent = @"maman";
printf("Je m'appelle %s, j'aime ma %s, elle est dans la %s...\n",
       nom, [parent UTF8String], [@"marine" UTF8String]);
NSLog(@"Je m'appelle %s, j'aime ma %@, elle est pas dans la %@
        en ce moment.\n", nom, parent, @"marine");
```

10 Fonctionnalités propres au C++

Jusque là, il a seulement été montré que les concepts objets du C++ étaient tous implémentés en Objective-C. Cependant, certaines particularités sont quant à elles purement absentes. Il ne s'agit cependant pas de concepts objets, mais bien de facilités de programmation.

10.1 Références

Les références (&) ne sont pas présentes en Objective-C. La gestion mémoire par compteur de référence et l'`autorelease` permettent de s'en passer. En fait, elles n'auraient pas grande utilité. Dans la mesure où les objets doivent toujours être alloués dynamiquement, on n'y fait référence que par pointeurs.

10.2 Inlining

L'inlining est également absent. Pour les méthodes, cela peut se comprendre, à cause du dynamisme d'Objective-C, qui ne peut se permettre de trop figer le code dans l'exécutable. En revanche, l'inlining peut faire défaut pour des fonctions C utilitaires, telles que `max()`, `min()`... C'est là un problème que peut résoudre Objective-C++.

Notez cependant que le compilateur GCC propose le mot-clef non standard `__inline` ou `__inline__` pour utiliser l'inlining en langage C, et donc en Objective-C. De plus, GCC sait également compiler le C99, une révision du langage C qui implémente l'inlining avec le mot-clef (standard cette fois) `inline`. L'Objective-C basé sur du C99 bénéficie donc également de l'inlining.

Il ne s'agit pas d'inlining, mais pour la recherche de performances, vous pouvez parfois vous tourner vers le IMP-caching (cf. section 11.3.2 page 58).

10.3 Templates

Les templates (ou *modèles*) sont un substitut au principe d'héritage et de méthodes virtuelles, conçu pour l'efficacité, mais intrinsèquement incompatible avec un modèle objet pur (Saviez-vous qu'un template permet de gagner un accès `public` sur des membres initialement `private` ?). Les templates ne sont pas implémentés en Objective-C, et ne pourraient l'être que difficilement compte tenu des règles de surcharge de fonctions.

Un lecteur attentif m'a toutefois fait part de développements en ce sens dans la librairie *MathArray* développée par *Adam Fedor*, que je cite ici pour les autres lecteurs intéressés.

10.4 Surcharge d'opérateurs

La surcharge d'opérateurs est absente d'Objective-C.

10.5 Friends

Il n'y pas de notion de *friend* en Objective-C. En effet, cette notion est surtout utile en C++ pour l'efficacité des opérateurs surchargés, qui ne sont pas disponibles en Objective-C. La notion de paquets Java, assez proches des *friends*, peut souvent être traitée par des catégories de classe (cf. Section 4.5 page 26).

10.6 Méthodes `const`

Les méthodes ne peuvent être déclarées `const` en Objective-C. Par conséquent, le mot-clef `mutable` n'a pas de raison d'être.

10.7 Liste d'initialisation dans le constructeur

Les listes d'initialisation à la construction n'existent pas en Objective-C.

11 STL et Cocoa

La bibliothèque standard du C++ est une autre de ses grandes forces. Même si elle a quelques lacunes, notamment dans les foncteurs (lacunes souvent comblées dans l'implémentation SGI de la STL [8]), elle est malgré tout très riche et vite indispensable. Ce n'est pas à proprement parler une partie du langage, puisqu'elle est construite dessus sans faire partie de sa grammaire, mais on a tôt fait d'en chercher un équivalent dans tout nouveau langage étudié. En l'occurence, en Objective-C, il faut bien sûr se tourner vers Cocoa pour trouver conteneurs, itérateurs et autres codes fournis clef en main.

11.1 Conteneurs

Bien sûr, l'approche Cocoa est purement objet : un conteneur n'est pas template, il ne peut contenir que des objets. Au moment où j'écris ces lignes, les conteneurs disponibles sont les suivants :

- `NSArray` et `NSMutableArray` pour les ensembles ordonnés ;
- `NSSet` et `NSMutableSet` pour les ensembles non ordonnés ;
- `NSDictionary` et `NSMutableDictionary` comme conteneurs associatifs ;
- `NSHashTable` comme table de hachage à références faibles (Objective-C 2.0 uniquement, voir plus bas).

Remarquez l'absence notable de classes comme « `NSList` » ou « `NSQueue` ». En effet, il semblerait que ces deux dernières classes n'aient intérêt à être implémentées que comme des `NSArray`. Cela peut sembler une absurdité du point de vue des performances, mais la réalité est autre.

En effet, contrairement à un `vector<T>` du C++, le `NSArray` de l'Objective-C encapsule réellement son implémentation et rend son contenu inaccessible autrement que par des appels de méthodes. Ainsi, rien n'oblige le `NSArray` à maintenir son contenu comme des cases mémoires contiguës. Les implémenteurs du `NSArray` ont vraisemblablement choisi une technique représentant un bon compromis pour l'utilisation du `NSArray` à la fois comme tableau et comme liste. N'oublions pas qu'en Objective-C, un conteneur ne contient que des pointeurs d'objets, ce qui permet d'assurer une manipulation très efficace des « cases ».

`NSHashTable` est équivalent à `NSSet`, mais utilise des références faibles (cf. section 6.6.2 page 50), ce qui est utile lors de l'utilisation du ramasse-miettes (section 6.6 page 50).

11.2 Itérateurs

11.2.1 Énumération classique

L'approche objet pure rend la notion d'itérateur plus souple qu'en C++. La classe `NSEnumerator` joue ce rôle. Exemple :

```
NSArray* tab = [NSArray arrayWithObjects:objet1, objet2, objet3, nil];
NSEnumerator* enumerateur = [tab objectEnumerator];
id unObjet = [enumerateur nextObject];
NSString* uneChaine = @"poufpouf";
while (unObjet != nil)
{
  [unObjet faireQuelqueChoseAvec:uneChaine];
  unObjet = [enumerateur nextObject];
}
```

Une méthode du conteneur (`objectEnumerator`) renvoie un itérateur, lequel est alors capable de se déplacer de lui-même (`nextObject`). Le comportement est plus proche de celui du Java que du C++. Lorsque l'itérateur a parcouru tout le conteneur, `nextObject` renvoie `nil`.

Il existe une forme plus courante car plus concise de l'utilisation des itérateurs, basée uniquement sur les possibilités syntaxiques du C.

```
NSArray* tab = [NSArray arrayWithObjects:objet1, objet2, objet3, nil];
NSEnumerator* enumerateur = [tab objectEnumerator];
id unObjet = nil;
NSString* uneChaine = @"poufpouf";
while ((unObjet = [enumerateur nextObject])) { //les double parenthèses évitent
  [unObjet faireQuelqueChoseAvec:uneChaine];   //un avertissement de gcc
}
```

11.2.2 Énumération rapide

Objective-C 2.0 (cf. section 1.3 page 6) a introduit une nouvelle syntaxe pour énumérer un conteneur, de telle sorte que le `NSEnumerator` soit implicite (il ne s'agit d'ailleurs plus d'un `NSEnumerator`). La syntaxe est de la forme :

```
NSArray* unConteneur = ...;
for(id objet in unConteneur) { //ici, chaque objet est typé "id"
  ...
}
for(NSString* objet in unConteneur) { //ici, chaque objet est typé "NSString*"
  ... //si un objet n'est pas une NSString*, c'est au développeur de le gérer
}
```

11.3 Foncteurs (objets-fonctions)

11.3.1 Utilisation du `selector`

La puissance du `selector` en Objective-C permet de se passer d'une notion spécifique de foncteur. En effet, le typage faible permet d'envoyer un message sans vraiment se soucier du récepteur. Par exemple, voici un code équivalent à celui présenté dans la section précédente sur les itérateurs :

```
NSArray* tab = [NSArray arrayWithObjects:objet1, objet2, objet3, nil];
NSString* chaine = @"poufpouf";
[tab makeObjectsPerformSelector:@selector(faireQuelqueChoseAvec:)
                     withObject:chaine];
```

Dans ce cas, rien n'oblige les différents objets à être de la même classe, ni même d'implémenter la méthode `faireQuelqueChoseAvec`: (au risque de faire lever une exception de « sélecteur non reconnu ») !

11.3.2 IMP caching

Cela ne sera pas détaillé ici, mais il est possible d'obtenir l'adresse réelle d'une fonction C représentant une méthode. Cela peut être utile pour optimiser de multiples appels au même `selector` sur un lot d'objets, puisque la résolution de méthode n'est alors effectuée qu'une fois. Cette technique est nommée « IMP caching » puisque `IMP` est en Objective-C le type de données pour une implémentation de méthode.

Un appel à `class_getMethodImplementation()` (cf. section 13.2 page 66) permet par exemple d'obtenir un tel pointeur. Notez cependant qu'il s'agit réellement d'un pointeur vers la fonction concernée, et qu'il ne résout pas les appels virtuels. Son utilisation relève donc le plus souvent de l'optimisation, et doit être employée avec prudence.

11.4 Algorithmes

Les très nombreux algorithmes généraux de la STL n'ont souvent pas de réel équivalent en Cocoa. Il faut plutôt se tourner directement vers les méthodes offertes par chaque conteneur.

12 Code implicite

Dans cette partie sont regroupées deux fonctionnalités dont la particularité est de permettre un allègement du code. Leurs vocations sont différentes : le *Key-value coding* (section 12.1) peut résoudre un appel de méthode indirect en utilisant la première implémentation valide trouvée, tandis que les propriétés (cf. section 12.2 page 61) permettent de faire générer certaines méthodes rébarbatives au compilateur.

Le *Key-value coding* est en réalité un service rendu par la richesse de Cocoa, tandis que la notion de *propriété* relève du langage lui-même, et a été ajoutée à Objective-C 2.0.

12.1 Key-value coding

12.1.1 Principe

Le *Key-value coding* est le nom que l'on donne à la pratique consistant à obtenir ou définir la valeur d'une donnée membre d'objet en y faisant référence par son nom. Cette pratique est similaire à l'utilisation d'un tableau associatif (`NSDictionary`, cf. section 11.1 page 57), le nom de la donnée membre faisant office de clef. La classe `NSObject` propose ainsi les méthodes `valueForKey:` et `setValue:forKey:`. Si les données membres sont elles-mêmes des objets, l'exploration peut être faite en profondeur, et la clef doit alors être un « keypath », dont les composantes sont séparées par des points. Les méthodes à utiliser sont alors `valueForKeyPath:` et `setValue:forKeyPath:`.

```
@interface A {
  NSString* toto;
}

... //il faut implémenter certaines méthodes pour que la suite fonctionne

@end

@interface B {
  NSString* titi;
  A* myA;
}

... //il faut implémenter certaines méthodes pour que la suite fonctionne

@end

@implementation B
...
//supposons un objet a de type A et un objet b de type B.
B* a = ...;
B* b = ...;
NSString* s1 = [a valueForKey:@"toto"]; //ok
NSString* s2 = [b valueForKey:@"titi"]; //ok
NSString* s3 = [b valueForKey:@"myA"]; //ok
NSString* s4 = [b valueForKeyPath:@"myA.toto"]; //ok
NSString* s5 = [b valueForKey:@"myA.toto"]; //erreur !
NSString* s6 = [b valueForKeyPath:@"titi"]; //ok, pourquoi pas
...
@end
```

Grâce à cette syntaxe, il est par exemple possible d'utiliser le même code pour manipuler des objets de classes différentes mais possédant des données aux noms identiques.

La véritable utilité réside dans la capacité à lier à une donnée (son nom) à des traitements particuliers (des appels de méthode notamment), comme au sein du *Key-Value Observing* (KVO), qui n'est pas détaillé ici.

12.1.2 Interception

L'accès à une donnée *via* un appel à `valueForKey:` ou `setValue:forKey:` n'est pas une opération atomique. Cet accès respecte une convention d'appel de méthode. En réalité, l'accès n'est possible que si certaines méthodes ont été implémentées (génération qui peut être implicite dans le cas des *propriétés*, cf. section 12.2 page suivante), ou qu'un accès direct aux variables d'instances a été explicitement autorisé.

La documentation d'Apple décrit très exactement le comportement de `valueForKey:` et de `setValue:forKey:` [3].

Pour un appel à `valueForKey:@"toto"`

- si elle existe, appel à la méthode `getToto` ;
- sinon, si elle existe, appel à la méthode `toto` (cas le plus courant) ;
- sinon, si elle existe, appel à la méthode `isToto` (assez courant pour les données booléennes) ;
- sinon, si la classe retourne `YES` pour la méthode `accessInstanceVariablesDirectly`, tente de lire la donnée membre (si elle existe) `_toto`, sinon `_isToto`, sinon `toto`, sinon `isToto`
- en cas de succès d'une des étapes précédente, la valeur trouvée est retournée ;
- en cas d'échec, la méthode `valueForUndefinedKey:` est appelée ; elle a une implémentation par défaut dans `NSObject`, qui lève une exception.

Pour un appel à `setValue:..forKey:@"toto"`

- si elle existe, appel à la méthode `setToto:` ;
- sinon, si la classe retourne `YES` pour la méthode `accessInstanceVariablesDirectly`, tente de modifier la donnée membre (si elle existe) `_toto`, sinon `_isToto`, sinon `toto`, sinon `isToto` ;
- en cas d'échec, la méthode `setValue:forUndefinedKey:`, implémentée par défaut dans `NSObject`, est appelée.

Notez bien ceci : **Un appel à `valueForKey:` ou `setValue:forKey:` peut servir à déclencher n'importe quelle méthode compatible ; il n'y a donc pas forcément de donnée membre sous-jacente, elle peut être « factice »**. Par exemple, appeler `valueForKey:@"length"` sur une chaîne de caractères est sémantiquement équivalent à appeler directement la méthode `length`, puisque c'est celle-là qui sera trouvée en premier dans le schéma de résolution du KVC.

Attention toutefois : les performances du KVC sont évidemment diminuées par rapport à un appel direct de méthode, il doit donc être utilisé à bon escient.

12.1.3 Prototypes

Notez que ce système nécessite tout de même de respecter le prototype attendu des méthodes pouvant être appelées : les accesseurs en lecture n'ont pas de paramètre et renvoient un objet, tandis que les accesseurs en écriture prennent un objet en paramètre et ne renvoient rien. Le type exact du paramètre objet n'a pas d'importance dans le prototype, puisqu'il est de type `id`.

Notez que les structures et les types de base (`int`, `float`...) sont supportés : le run-time Objective-C est capable d'effectuer un *boxing* automatique (une encapsulation) dans un objet `NSNumber` ou `NSValue`. Le retour de `valueForKey:` est donc toujours un objet.

Le cas particulier de la valeur `nil` passée à `setValue:forKey:` est traité par une méthode `setNilValueForKey:`.

12.1.4 Fonctionnalités avancées

Il subsiste encore quelques détails dignes d'intérêt, mais qui ne sont pas détaillés ici.

- Le premier concerne les *keypath* qui peuvent inclure des traitements particuliers, comme un calcul de somme, de moyenne, de valeur max ou min...Le caractère `@` en est le signe distinctif.
- Le second concerne la cohérence que doit avoir un appel à `valueForKey:` ou `setValue:forKey:` par rapport aux méthodes `objectForKey:` et `setObject:forKey:` que peuvent présenter les collections comme les tableaux associatifs (cf. section 11.1 page 57). Là encore, le caractère `@` est utilisé, pour lever certaines ambiguïtés.

12.2 Propriétés

12.2.1 Utilité des propriétés

La notion de *propriété* se rencontre à la déclaration des classes. On peut associer le mot-clef `@property` (ainsi que quelques attributs, cf. section 12.2.3 page suivante) à une donnée membre, pour préciser que ses accesseurs doivent être, ou non, générés automatiquement par le compilateur, et si oui, comment. Le but est une économie de code, et de temps de développement.

De plus, la syntaxe permettant d'accéder aux propriétés est plus légère qu'un appel de méthode, il peut donc être agréable d'utiliser une propriété, même si l'on écrit les accesseurs soi-même. Les performances, elles, sont identiques, car c'est la compilation qui résout l'appel de méthode sous-jacent.

La plupart du temps, une propriété est liée à une donnée membre. Mais si les accesseurs sont redéfinis, rien n'empêche la propriété d'être « factice », autrement dit, elle peut être semblable de l'extérieur à un attribut de l'objet, mais avec un comportement plus complexe en interne que le simple changement de valeur d'une donnée membre.

12.2.2 Description des propriétés

Décrire une propriété nécessite de préciser les directives d'implémentation que doivent suivre les accesseurs :

- est-ce une propriété en lecture seule pour l'extérieur ?
- si la donnée membre correspondante est un type de base, il n'y a pas tellement de choix, mais s'il s'agit d'un objet, doit-il être encapsulé par copie, par référence forte, ou par référence faible ? (cela est lié à la gestion mémoire, cf. section 6.4.7 page 45) ;
- faut-il rendre les accès *thread-safe* (cf. section 8.1 page 54) ?
- quel est le nom des accesseurs ?
- à quelle donnée d'instance la propriété doit-elle être reliée ?
- quels accesseurs ne faut-il pas générer automatiquement, pour laisser le soin au développeur de les implémenter lui-même ?

Répondre à ces questions se fait en deux étapes :

- dans le bloc `@interface` d'une classe, on déclare les propriétés, nanties des attributs adaptés (cf. section 12.2.3 page suivante) ;
- dans le bloc `@implementation` de cette même classe, on précise que les accesseurs sont implicites, ou on en fournit une implémentation (cf section 12.2.4 page 63).

Le prototype des accesseurs est imposé : pour un accesseur en lecture, il renvoie le type attendu (ou compatible), et pour l'écriture, il renvoie `void` et prend un seul paramètre du type attendu (ou compatible).

Le nom des accesseurs est codifié également : pour une variable `toto`, les noms par défaut sont `toto` pour la lecture, et `setToto`: pour l'écriture. Il est possible de spécifier d'autre noms. Notez cependant que contrairement au cas du *Key-Value Coding* (section 12.1.2 page précédente), le nom doit être connu **à la compilation**, car l'utilisation des *propriétés* doit être aussi performant que d'utiliser les méthodes associées. De même, aucun *boxing* n'est appliqué aux types incompatibles passés en paramètre.

Ci-après vient un exemple avec peu d'explications, mais qui permet de visualiser le fonctionnement global. Les sous-sections suivantes donnent les détails nécessaires à la compréhension.

```
@interface class Voiture : NSObject
{
  NSString* immatriculation;
  Personne* conducteur;
}

//l'immatriculation est en lecture seule, et initialisée par copie
@property NSString*  (readonly, copy) immatriculation;

//le conducteur est une référence faible (aucun retain), et peut être modifié
@property Personne* (assign) conducteur;

@end

...

@implementation

//laisser le compilateur générer le code pour "immatriculation" si le
//développeur  ne le fait pas lui-même
@synthesize immatriculation;

//le développeur fournit l'implémentation des accesseurs pour "conducteur"
@dynamic conducteur;

//cette méthode convient pour l'accesseur en lecture de @dynamic conducteur
-(Personne*) conducteur  {
 ...
}

//cette méthode convient pour l'accesseur en écriture de @dynamic conducteur
-(void) setConducteur:(Personne*)valeur {
 ...
}

@end
```

12.2.3 Attributs des propriétés

Une propriété est déclarée selon le modèle suivant :
`@property` *type nom*;
ou
`@property(`*attributs*`)` *type nom*;

S'ils ne sont pas précisés, les attributs ont une configuration par défaut ; sinon, ils peuvent être redéfinis pour répondre aux question posées dans la section précédente. Ils peuvent être :

- `readwrite` (par défaut) ou `readonly` pour spécifier si la propriété doit avoir les deux accesseurs (lecture/écriture) ou seulement en lecture ;
- `assign` (par défaut), `retain` ou `copy`, pour spécifier la façon dont est stockée la valeur en interne ;
- `nonatomic` pour ne pas générer de code `thread-safe`, comme le ferait le comportement par défaut (il n'y a pas de mot-clef `atomic`) ;
- `getter=..., setter=...` pour changer le nom choisi par défaut pour les accesseurs.

Au sein de l'accesseur en écriture, les comportements `assign`, `retain` ou `copy` changent la façon dont est initialisée la donnée d'instance.

Dans une méthode `-(void) setToto:(Toto*)valeur`, les trois variantes correspondantes sont :

```
self->toto = valeur; //assignation simple
self->toto = [valeur retain]; //assignation avec incrément du compteur de référence
self->toto = [valeur copy]; //l'objet est copié ; il doit respecter le protocole
                            //NSCopying (cf. section 5.3.1 page 36
```

Notez que dans le cas où le ramasse-miettes est activé (cf. section 6.6 page 50), `retain` ne change rien par rapport à `assign`. Mais dans ce cas, sachez aussi que les attributs `__weak` et `__strong` peuvent être spécifiés.

```
@property(copy,getter=getS,setter=setF:) __weak NSString* s; //déclaration complexe
```

(notez la syntaxe « setF : » **avec les deux-points**)

12.2.4 Implémentations personnalisées des propriétés

L'exemple de code de la section 12.2.2 page 61 montre que l'implémentation repose sur simplement deux mots-clefs : `@synthesize` et `@dynamic`.

`@dynamic` stipule que c'est le développeur qui fournit les implémentations nécessaires (un simple accesseur en lecture si *read-only* a été spécifié lors de la déclaration de la propriété ; les deux accesseurs sinon).

`@synthesize` stipule que, *à moins que le développeur ne l'ai déjà fait*, le compilateur doit générer lui-même les accesseurs, en respectant les contraintes de la déclaration. Ainsi, dans l'exemple donné, si le développeur avait implémenté une méthode `-(NSString*)immatriculation`, le compilateur aurait choisi celle-là et n'en aurait pas généré d'autre. De cela, on déduit aussi qu'un des deux accesseurs peut être généré automatiquement, et l'autre fourni par le programmeur.

Notez enfin que si un accesseur n'était pas présent lors de la compilation, et n'a pas non plus été créé par le compilateur par `@synthesize`, rien n'empêche de le rajouter au run-time (cf. section 13.2 page 66). Cela est valide pour l'accès la propriété. Mais dans ce cas, le nom de l'accesseur est bien choisi lors de la compilation.

Si, au run-time, aucun accesseur n'est disponible, une exception est levée, mais le programme ne s'interrompt pas, tout comme pour un simple appel de méthode.

Dans le cas du `@synthesize`, on peut demander au compilateur de lier la propriété à une donnée membre particulière, qui n'a pas forcément le même nom que la propriété.

```
@interface A : NSObject {
  int _toto;
}
@property int toto;
@end

@implementation A
@synthesize toto=_toto; //lier à "_toto" plutôt que "toto"
                        //(qui n'existe d'ailleurs pas ici)
@end
```

12.2.5 Syntaxe des accès aux propriétés

Pour obtenir ou définir la valeur d'une propriété, la syntaxe utilise le point : c'est donc la même syntaxe qu'une simple `struct` en C, et reste cohérent avec le principe des *keypath* (cf. section 12.1.1 page 59). La performance à l'exécution est identique à l'appel des accesseurs sous-jacent.

```
@interface A : NSObject {
  int i;
}
@property int i;
@end

@interface B : NSObject {
  A* myA;
}
@property(retain) A* a;
@end

...
A* a = ...
B* b = ...;
a.i = 1; //équivalent à [a setI:1];
b.myA.i = 1;//équivalent à [[b myA] setI:1];
```

Notez qu'au sein d'une méthode de la classe `A` de l'exemple précédent, la différence serait grande entre `self->i` et `self.i`. En effet, `self->i` est un accès direct à la donnée d'instance, tandis que `self.i` déclenche le mécanisme d'accès à la propriété, et donc un appel de méthode.

12.2.6 Détails avancés

La documentation des propriétés [4] stipule que dans une compilation 64-bits, le run-time Objective-C apporte quelques différences par rapport au mode 32-bits. Les données membres correspondant aux déclarations `@property` peuvent par exemple être omises, car implicites. La documentation d'Apple reste donc incontournable pour l'exhaustivité des renseignements.

13 Dynamisme

13.1 RTTI (Run-Time Type Information)

Le langage C++ est en quelque sorte un faux langage objet ; comparé à Objective-C, il est extrêmement statique. C'est un choix tourné délibérément vers la recherche de performances d'exécution maximales. Les informations que l'on peut obtenir pendant l'exécution du programme, en C++, *via* la bibliothèque *typeinfo*, ne sont pas très fiables, car elles dépendent du compilateur. Connaître la classe d'un objet est une question que l'on se pose rarement, à cause du typage fortement statique. Mais elle peut être soulevée lors de l'exploration d'un conteneur. Le `dynamic_cast` et, plus rare, le `typeid`, peuvent être utilisés, mais ne permettent pas une grande interaction avec l'utilisateur du programme. Comment tester efficacement si un objet est une instance d'une classe donnée par l'utilisateur sous forme de chaîne de caractères ?

Objective-C est armé, par nature, pour ce genre de questions. Les classes étant des objets, elles en héritent des comportements.

Remarquez que le *downcasting* a été traité en section 3.4.4 page 21 car il s'agit d'un usage particulier du `dynamic_cast`.

13.1.1 `class`, `superclass`, `isMemberOfClass`, `isKindOfClass`

L'aptitude d'un objet à connaître son type lors de l'exécution du programme est appelée *introspection*, et s'appréhende par différentes méthodes.

`isMemberOfClass`: est une méthode qui répond à la question : « suis-je une instance d'une classe donnée (sans considérer l'héritage) ? », tandis que `isKindOfClass`: répond à « suis-je une instance d'une classe donnée ou d'une dérivation de cette classe ? ».

L'utilisation de ces méthodes nécessite d'utiliser le faux mot-clef : `class` (et non `@class`, utilisé en déclaration forward). En effet, `class` est une méthode de NSObject, et renvoie un objet `Class`, qui est l'*objet de classe* (l'instance de la méta-classe). Notez au passage que la valeur « vide » d'un pointeur de classe n'est pas `nil` mais `Nil`.

```
BOOL test = [self isKindOfClass:[Toto class]];
if (test)
  printf("Je suis dans la classe de Toto\n");
```

Notez qu'il existe également un moyen d'obtenir l'objet de classe de la classe mère : on peut utiliser pour cela la méthode `superclass`.

13.1.2 `conformsToProtocol`

Cette méthode a été introduite dans la section consacrée aux protocoles (section 4.4 page 23). Elle permet de savoir si un objet adhère à un protocole ou non. Ce n'est pas extrêmement dynamique, car le compilateur se base sur ce qui a été déterminé explicitement dans le code source. Si un objet implémente toutes les méthodes d'un protocole, mais sans le déclarer officiellement, le programme est correct, mais `conformsToProtocol`: renvoie `NO`.

13.1.3 `respondsToSelector`, `instancesRespondToSelector`

`respondsToSelector`: est une méthode d'instance, héritée de `NSObject`. Elle est capable de déterminer si un objet implémente une méthode donnée, qu'elle soit héritée ou non. On utilise pour cela la notion de *sélecteur* (cf. section 3.3.4 page 15). Exemple :

```
if ( [self respondsToSelector:@selector(travailler)] )
{
  printf("Je ne suis pas Toto.\n");
  [self travailler];
}
```

Pour savoir si une classe donnée implémente une méthode, sans considérer l'héritage, on dispose de la méthode *de classe* `instancesRespondToSelector:`.

```
if ([[self class] instancesRespondToSelector:@selector(trouverBoulot)])
{
    printf("Je suis capable de trouver un boulot"
           " sans l'aide de ma maman.\n");
}
```

Notez qu'un appel à `respondsToSelector:` ne peut déterminer si une classe reconnaît un message par *forwarding* (cf. section 3.4.3 page 20).

13.1.4 Typage fort ou typage faible *via* `id`

Le C++ est un langage au typage fort : on ne peut utiliser un objet que si l'on connaît son type. En Objective-C, la contrainte est moins forte : si un objet **au type explicite** est la cible d'un message qu'il ne sait pas traiter *a priori*, le compilateur génère un avertissement (*warning*) mais le programme fonctionnera. Le message sera simplement perdu (en levant une exception), sauf si par *forwarding* (cf. section 3.4.3 page 20) il peut être récupéré. Si c'est bien ce qui a été prévu par le développeur, l'avertissement est une fausse alerte ; dans ce cas on l'évite en donnant à la cible du message le type `id` plutôt que son type réel. En fait, tout objet est de type `id`, et peut sous ce typage être la cible de n'importe quel message.

Cette fonctionnalité de typage faible est indispensable lorsque l'on utilise des processus de délégation : il n'est pas besoin de connaître le type exact d'un objet délégué pour pouvoir l'utiliser. En voici un exemple :

```
-(void) setAssistant:(id)unObjet
{
  [assistant autorelease];
  assistant = [unObjet retain];
}

-(void) traiterLeDossier:(Dossier*)dossier
{
  if ([assistant respondToSelector:@(traiterleDossier:)])
    [assistant traiterLeDossier:dossier];
  else
    printf("Vous avez le formulaire bleu ?\n");
}
```

Précisons enfin que les délégations sont monnaie courante en Cocoa dès que l'on utilise une interface graphique, dans laquelle tous les contrôles sont chargés de transmettre des actions de la part de l'utilisateur.

13.2 Manipulation des classes Objective-C durant l'exécution

En incluant le fichier d'en-tête `<objc/objc-runtime.h>`, on a accès à de nombreuses fonctions utilitaires pour modifier les informations des classes, comme les méthodes ou les variables d'instance, durant l'exécution du programme.

Objective-C 2.0 a introduit de nouvelles fonctions, plus pratiques que les précédentes (comme `class_addMethod(...)` au lieu de `class_addMethods(...)`), rendant obsolètes la plupart des fonctions de la version 1.0.

Il est ainsi remarquablement facile de modifier les classes durant l'exécution du programme. Bien que ce ne soit pas une pratique courante, il existe des cas où cette facilité est un atout extrêmement précieux.

14 Objective-C++

Le langage Objective-C++ est en cours de développement. Il est déjà fonctionnel, mais présente encore quelques lacunes. Objective-C++ permet d'utiliser conjointement de l'Objective-C et du C++, pour tirer parti des fonctionnalités de chacun.

Sur des points comme la gestion conjointe des exceptions C++ et Objective-C, ou l'utilisation d'objets C++ comme données d'instance de classes Objective-C, la documentation doit être consultée pour connaître le comportement normal attendu. Des programmes peuvent cependant d'ores et déjà être écrits en Objective-C++. Les fichiers d'implémentation portent l'extension *.mm*.

Depuis MacOS X 10.4, un objet C++ peut être utilisé comme donnée d'instance d'une classe Objective-C, et bénéficier d'une construction/destruction automatique [3].

15 Évolutions d'Objective-C

Je n'ai aucune information sur le développement futur d'Objective-C, en version 3.0 ou autre chose. Cependant, Objective-C bénéficie automatiquement des améliorations apportées au langage C. Les projets GCC [4] et LLVM [5] sont à ce sujet très intéressant à suivre.

GCC, pour suivre les standards, améliore pas à pas son support du C++0x. Objective-C++pourra donc certainement en bénéficier. De plus, LLVM s'est récemment vu doté du support des *closures* par l'intermédiaire des `blocks` (cf. 15.1). Ici, c'est le le langage C qui reçoit une évolution majeure, par conséquent l'Objective-C également.

Le langage est donc toujours en développement, directement ou indirectement, et laisse augurer du meilleur sur son évolution.

15.1 Les *blocks*

15.1.1 Support et utilisation

Depuis MacOS X 10.6, les API natives, comme celles de Grand Central Dispatch, font parfois usage des *blocks*, avec une syntaxe ressemblant aux pointers de fonction, l'étoile étant remplacée par un accent circonflexe. Un exemple :

```
void dispatch_apply(
  size_t iterations, dispatch_queue_t queue, void (^block)(size_t));
```

Le dernier paramètre n'est pas un pointeur de fonction, mais un *block*.

Un *block* est une notion qui n'est liée ni à Cocoa, ni à l'Objective-C. Il s'agit d'une fonctionnalité du langage C, non standard, non supportée par tous les compilateurs. Le compilateur LLVM livré avec MacOS X 10.6 le supporte.

Pour les connaisseurs, il s'agit de l'implémentation en C des *closures*, déjà utilisées en Javascript. Pour simplifier, cela peut être vu comme des fonctions anonymes, créées à la volée, capturant leur environnement (valeurs des variables) lors de leur création.

Les blocks sont très pratiques pour remplacer les callbacks, et en règle générale diminuer la quantité de code à écrire pour les appels de fonction retardés ou asynchrones.

3. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC`
4. `http://gcc.gnu.org/`
5. `http://llvm.org/`

15.1.2 Syntaxe

Un block est créé par la syntaxe

```
^{...du code...}
```

ou encore

```
^(paramètre1,[autres paramètres...]){...du code...}
```

Il peut être stocké dans une variable et appelé comme un pointeur de fonction, où l'accent circonflexe remplace l'étoile.

```
typedef void (^b1_t)(void);
b1_t block1 = ^{printf("coucou\n");};
block1(); //affiche coucou

typedef void (^b2_t)(int i);
b2_t block2 = ^(int i){printf("%d\n", i);};
block2(3);//affiche 3
```

15.1.3 Capture de l'environnement

Un block peut référencer les variables existantes au moment de sa création. Il « capture » l'état des variables, et peut ainsi les utiliser, **en lecture seule**. Modifier une variable extérieure à un block requiert des instructions spécifiques (cf. section 15.1.4).

```
//exemple 1
int i = 1;
b1_t block1 = ^{printf("%d", i);};
block1();//Affiche 1
i = 2;
block1();//Affiche toujours 1 ! La valeur utilisée est celle qui a été capturée

//exemple 2
int i = 1;
b1_t block1 = ^{i=2;}//invalide : ne compile pas.
                     //La variable ne peut être modifiée ainsi.

//exemple 3
typedef void (^b1_t)(void);
b1_t getBlock() //cette fonction renvoie un block
{
  int i = 1; //cette variable est locale à la fonction
  b1_t block1 = ^{printf("%d", i);};
  return block1;
}

b1_t block1 = getBlock();
block1(); //affiche 1 : le block a capturé la variable locale lorsqu'elle était
          //dans sa portée.
```

15.1.4 Variables `__block`

Le fait qu'un block semble être capable d'utiliser une variable même lorsqu'elle a disparu de la portée du code en cours d'exécution semble trivial : il n'a fait qu'en créer une copie. C'est pourquoi, pour éviter les confusions, il n'est pas possible de modifier la valeur d'une telle variable dans un block.

Il est toutefois possible de lever cette limitation. Si une variable est déclarée avec l'attribut `__block`, alors un block peut la modifier. Cela est plus délicat, car cette fois, la variable capturée doit toujours être dans la portée du block lorsque celui-ci est appelé.

Veuillez noter qu'une variable `extern` ou `static`, pusiqu'elle n'est pas `auto` (le cas par défaut en langage C), peut être modifiée par un block.

```
//exemple 1
__block int i = 1;
b1_t block1 = ^{printf("%d", i);};
block1();//Affiche 1
i = 2;
block1();//Affiche 2

//exemple 2
__block int i = 1;
b1_t block1 = ^{printf("++i = %d", ++i);};
block1();//Affiche 2
block1();//Affiche 3

//exemple 3
typedef void (^b1_t)(void);
b1_t getBlock() //cette fonction renvoie un block
{
  __block int i = 1; //cette variable est locale à la fonction, le block renvoyé
                     //ne pourra donc être valide hors de getBlock();
  b1_t block1 = ^{printf("%d", i);};
  return block1;
}

b1_t block1 = getBlock();
printf("Attention, appel suivant incorrect:\n");
block1(); //probablement Segmentation Fault à cause de la variable "i"
```

Conclusion

Ce document n'est qu'un aperçu rapide des différents aspects d'Objective-C comparé au C++. Il est cependant utile en tant que référence rapide pour les développeurs confirmés désirant comprendre ce langage. En espérant que le but soit atteint au mieux, j'invite cependant le lecteur avisé à me faire part de toute remarque, critique ou correction, pour améliorer tout ce qui peut l'être.

Références

[1] Apple Computer, Inc. Autozone. `http://www.opensource.apple.com/darwinsource/10.5.5/autozone-77.1/README.html`.

[2] Apple Computer, Inc., Developer documentation. Garbage Collection Programming Guide. `http://developer.apple.com/documentation/Cocoa/Conceptual/GarbageCollection.pdf`.

[3] Apple Computer, Inc., Developer documentation. Key-Value Coding Programming Guide. `http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding/KeyValueCoding.pdf`.

[4] Apple Computer, Inc., Developer documentation. The Objective-C 2.0 Programming Language. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[5] Boost. `http://www.boost.org/`.

[6] GNUstep. `http://www.gnustep.org/`.

[7] Aaron Hillegass. *Cocoa Programming for MacOS X, 2nd edition.* Addison-Wesley, 2004.

[8] SGI. Standard template library programmer's guide. `http://www.sgi.com/tech/stl`.

[9] Will Shipley. `self = [supid init]`. `http://wilshipley.com/blog/2005/07/self-stupid-init.html`.

Révisions du document

version 2.1
- ajout d'explications pour les *blocks*.

version 2.0
- mise à jour pour Objective-C 2.0 ;
- nombreuses retouches.

version 1.11
- correction de coquilles.

version 1.10
- adaption en anglais par Aaron VEGH ;
- réécriture de la section sur le clonage ;
- ajout d'une section sur le IMP caching ;
- précisions sur `initialize` ;
- précisions sur Objective-C++ ;
- correction des propos sur `superclass` ;
- précisions sur les données « static »de classe ;
- précisions sur les étiquettes des paramètres de méthodes ;
- correction de coquilles sur les constructeurs virtuels ;
- correction d'erreurs à propos de `self` et `super` ;
- précisions sur Objective-C 2.0 ;
- précisions sur les chaînes statiques ;
- corrections de petites coquilles diverses.

version 1.9
- changements mineurs.

version 1.8
- précisions sur Objective-C 2.0 (ramasse-miettes, énumération rapide, *properties*) ;
- précisions sur l'envoi d'un message à `nil` ;
- précisions sur les cycles de retain ;
- précisions sur les mutateurs et accesseurs en lecture ;
- quelques reformulations et corrections typographiques.

version 1.7
- corrections de fautes de frappe ([release chaine] au lieu de [self->chaine release]).

version 1.6
- corrections de fautes de frappe.

version 1.5
- un grand merci à Jean-Daniel Dupas pour ses corrections !
- correction : (partie 3.2.3) en Objective-C, l'attribut par défaut n'est pas `private` mais `protected` ;
- correction : (partie 3.3.5) `@selector()` est évalué à la compilation et ne pénalise pas les performances ;
- correction : (partie 3.4.2) un `retain` est plus logique qu'un `copy` ;
- correction : (parties 4.4.3 et 4.6) on ne peut pas utiliser catégorie et super-classe en même temps ;
- correction : (partie 5.1.1) la donnée d'instance `isa` est initialisée dans `alloc` et non dans `init` ;
- précisions sur *l'inlining* dans la partie 10.2 ;
- les méthodes cString et initWithCString sont devenues obsolètes avec MacOS 10.4, il faut maintenant utiliser les équivalents UTF8 ;
- précisions sur self = [super init] dans la partie 5.1.3 ; les différents code d'exemple utilisant `self = [super init] ; if (self){...} ;` ont été corrigés en conséquence ;
- ajout de la partie 5.1.5 intitulée « Échec de l'initialisation » ;
- légère remise en page (séparation des sections par un saut de page).

version 1.4
- Précisions sur l'absence de NSList, NSQueue ;
- précisions sur les itérateurs ;

– corrections typographiques.

version 1.3

– Ajout d'une partie sur les arguments anonymes (ou muets) ;
– ajout d'une partie sur les données de classe ;
– ajout d'une partie sur les accesseurs en lecture ;
– légère modification du nom de la section traitant des mutateurs ;
– précision sur la possibilité de nommer une méthode comme une donnée d'instance ;
– ajout de « méthodes virtuelles pures » dans l'index.

version 1.2

– La partie sur les mutateurs était incorrecte ! Elle est maintenant conforme au *Cocoa programming for MacOS X* [7] ;
– modification de la section « Appel de méthodes » en « Différenciation entre fonctions et méthodes » ; contenu légèrement modifié également ;
– correction de quelques erreurs typographiques et de fautes d'orthographe.

version 1.1

– Ajout de la présente section ;
– les renvois du document sont maintenant des hyperliens pour naviguer plus rapidement ;
– meilleure gestion des accents dans les lecteurs PDF ;
– explications plus complètes sur la copie (clonage) ;
– plus de détails sur les objets mutables ;
– référence au « Cocoa Programming for MacOS X » ;
– référence à SGI ;
– quelques petits remaniements de phrases.

Index