9

Vie et mort des objets

Ce chapitre a pour objectif de présenter les différentes manières d'effacer les objets de la mémoire pendant qu'un programme s'exécute. Nous verrons comment les langages utilisés dans ce livre traitent de ce problème : de la version libérale du C++, confiant la responsabilité au seul programmeur, aux versions plus « marxistes » de Java, Python et PHP 5, laissant un système de régulation centralisé extérieur, appelé ramasse-miettes, s'en occuper, en passant par la troisième voie chère à Tony Blair et proposée par le C#.

Sommaire : Gestion de la mémoire RAM — Dépenses de mémoire inhérentes à l'OO — Mémoire pile et mémoire tas — Le « delete » du C++ — Le ramasse-miettes de Java, C#, PHP 5 et Python

Candidus — Comment les objets s'arrangent-ils avec la mémoire ?

Doctus — Un objet est constitué d'un ensemble de données et de méthodes pour les manipuler. Lorsqu'il entre en scène un processus de chargement réserve la place nécessaire au stockage de ces deux ingrédients.

Cand. — Tu veux dire que les segments de code doivent également entrer dans les préoccupations du programmeur ! Tu parles d'un progrès !

Doc. — Bien que le code ne soit chargé qu'à un seul exemplaire, un objet est tout de même plus encombrant qu'une donnée primitive. Mais tu oublies une chose, il disposera des mécanismes nécessaires pour traiter la question. Sa suppression de la mémoire fait partie de son cycle de vie.

Cand. — Veux-tu dire que ça se fait tout seul ?

Doc. — En C++, non, mais en Java, C#, PHP 5 et Python, tu disposes de l'allocation et de la libération automatiques de mémoire.

Cand. — Je me contente donc d'appliquer les principes de localisation des données là où elles seront utilisées plutôt que de les allouer globalement au début du programme ?

Doc. — C'est bien ce que proposent ces quatre langages. Le mécanisme du ramasse-miettes, encore appelé *Garbage Collector*, prendra le soin de déterminer les circonstances où les données temporaires ont fini de servir et s'arrangera pour n'intervenir qu'en cas de réel besoin.

Cand. — Cela semble miraculeux… Comment fait ce ramasse-miettes pour savoir à coup sûr qu'une donnée ne sera plus utilisée ?

Doc. — Il n'y a aucune magie derrière tout ça ! Ces langages disposent d'un mécanisme pour détecter les occasions où le programme coupe les liens avec ses données temporaires. L'idée principale repose sur le fait que le seul moyen de créer un objet consiste à utiliser les zones mémoire contrôlées par la machine virtuelle et que cette même machine peut détecter que cet objet est devenu inutilisable et parfait pour la « casse ».

Question de mémoire

Un rappel sur la mémoire RAM

Object wanted : dead or alive ! Ce chapitre a la sinistre mais non impossible mission de vous expliquer le cycle de vie des objets, comment ils vécurent et comment ils sont morts. Vous en voulez encore ? Alors écoutez l'histoire de … Mais d'abord, quelques rappels élémentaires sur le fonctionnement d'un ordinateur lorsqu'il exécute un programme seront bienvenus en guise d'introduction. Un programme, pour qu'il s'exécute, nécessite, avant tout, de l'espace mémoire, pour pouvoir y stocker les données qu'il manipule et les instructions responsables de ces manipulations. Lors de l'exécution d'un programme OO, il faudra pouvoir stocker, et les objets et les méthodes.

La mémoire dite RAM, ou vive ou encore centrale, sert à cela. C'est une mémoire rapidement accessible, volatile et chère, au contraire du disque dur qui lui, le pauvre, est lent à la détente, mais permanent et à bon marché. De plus, elle doit se partager entre les multiples programmes qui peuvent s'exécuter en même temps, chacun ayant droit à sa part du gâteau. Les différents programmes auront une zone mémoire propre qui leur sera réservée, comme un casier dans un vestiaire, et qu'ils utiliseront exclusivement durant leur exécution. Aujourd'hui, on dit que les applications sont bien cloisonnées entre elles, ce qui permet d'éviter que l'une s'aventure dans un territoire réservé à l'autre, car, à l'instar des guerres de gangs à Los Angeles, cela peut faire beaucoup de dégâts.

Bien sûr, lorsque le programme s'interrompt, qu'il soit normalement ou anormalement terminé, toute la mémoire se vide, et c'est alors l'hécatombe du côté des objets. Et c'est bien pour cela qu'il faudra, si ce que ceux-ci sont devenus vous importe encore, vous préoccuper de sauver leur état, d'une manière ou d'une autre, sur le disque dur (nous aborderons la sauvegarde des objets sur le disque dur au chapitre 19).

Pour qu'un programme tourne vite, il est idéal que toutes les données et instructions qu'il manipule puissent être stockées dans la RAM, sinon le programme rame… Ce que l'on ne peut installer dans la RAM pourra, en dernier recours, être stocké sur le disque dur (on parle alors de mémoire virtuelle), provoquant en cela un effondrement des performances, vu que celui-ci prend pour l'extraction des données un million de fois plus de temps que la mémoire RAM. Cette mémoire-là est donc extrêmement précieuse mais, vu sa sophistication et son prix, non extensible à l'infini.

Par ailleurs, comme vous l'aurez constaté dans la pratique, en installant la nouvelle version de votre logiciel favori, plus on en a, plus on en use, pour ne pas dire abuse. La gourmandise (ou plutôt l'avidité des applications) s'adapte à la disponibilité des ressources. La mémoire RAM brûle les poches des développeurs d'application. De fait, une des préoccupations des programmeurs d'antan était d'économiser les ressources de l'ordinateur lors du développement des applications, le temps calcul et la mémoire. Aujourd'hui, l'existence même de pratique informatique comme l'OO permet de s'affranchir quelque peu de ce souci d'optimisation, pour le remplacer graduellement par un souci de simplicité, clarté, adaptabilité et facilité de maintenance. Ce que l'on gagne d'un côté, on le perd ailleurs. En effet, la pratique de l'OO ne regarde pas trop à la dépense, et ce, à plusieurs titres.

L'OO coûte cher en mémoire

L'objet, déjà en lui-même, est généralement plus coûteux en mémoire que les simples variables `int`, `char` ou `double` de type prédéfini. Il pousse à la dépense. De plus, rappelez-vous le « new », qui vous permet d'allouer de la mémoire pendant le déroulement de l'exécution du programme, et ce n'importe où. Alors pourquoi s'en priver ? À la différence d'autres langages, tout l'espace mémoire utilisé pendant l'exécution du programme n'est pas déterminé à l'avance, ni optimisé par l'étape de compilation.

Par ailleurs, certains langages OO, et non des moindres comme C++, sont des grands consommateurs d'objets temporaires utilisés, ou dans le passage d'argument ou comme variable locale (nous reviendrons sur ce point précis dans la suite). Bien que la pratique de l'OO soit une grande consommatrice de mémoire RAM, et que celle-ci va s'accroissant dans les ordinateurs suivant la fameuse loi de Moore (qui, comme un 11e commandement à force d'être citée, dit que tout en informatique fait l'objet d'un doublement de capacité tous les 16 mois), elle reste une ressource extrêmement précieuse, et toute pratique visant à économiser cette ressource pendant l'exécution du programme est plus qu'appréciable.

Économiser de la mémoire

La mémoire RAM est une denrée rare et chère, qu'il est important de gérer au mieux pendant l'exécution du programme, au risque de déborder sur le disque dur, avec, pour conséquence, un effondrement des performances.

Qui se ressemble s'assemble : le principe de localité

Un autre point capital dans la gestion de la mémoire est qu'il est important que les instructions et les données qui seront lues et exécutées à la suite se trouvent localisées dans une même zone mémoire. La raison en est l'existence aujourd'hui dans les ordinateurs d'un système de mémoire hiérarchisé (telle la mémoire cache), où des blocs de données et d'instructions sont extraits d'un premier niveau lent, pour être installé dans un second niveau plus rapide. Cela permet, lors de l'exécution du programme, d'extraire, le plus souvent possible, les données nécessaires à cette exécution hors du premier niveau.

Suite aux ratés, quand ce qui est requis pour la poursuite de l'exécution ne se trouve plus dans le niveau rapide, il sera nécessaire d'extirper à nouveau un bloc de données du niveau lent, en ralentissant considérablement l'exécution. Si lors du transfert de la mémoire lente vers la mémoire rapide, on ramène un peu plus que le strict nécessaire, et au vu du principe de localité, alors la probabilité d'un raté sera diminuée d'autant, car il y a de fortes chances que le surplus du transfert réponde aux prochaines requêtes. Comme les objets, au fur et à mesure de leur création, peuvent s'installer n'importe où dans la mémoire, et que l'essentiel de l'exécution consiste à passer d'un objet à l'autre, on conçoit que, là encore, la pratique OO soit presque antinomique avec toutes les démarches d'économie et d'accélération des performances. On verra qu'afin de diminuer les effets néfastes d'une telle répartition des objets, des systèmes automatiques cherchent à compacter au mieux la zone mémoire occupée par ces objets, et à les maintenir le plus possible dans la mémoire cache.

Les objets intérimaires

Si, au fur et à mesure de son exécution, le programme rajoute de nouveaux objets dans la mémoire, il serait commode, dans le même temps, de se débarrasser de ceux devenus inutiles et encombrants. Mais quand un objet devient-il inutile ? Tout d'abord, quand le rôle qu'il doit jouer est par essence temporaire. Par exemple, quand il permet à des structures de données de se transformer en passant par lui, mais n'est plus requis une fois les structures finales obtenues. En Java, existe la classe `Integer` qui permet, entre autres, de créer des

objets entiers à partir de String (chaîne de caractères), et de les manipuler, pour, une fois ces manipulations terminées, les stocker dans une simple variable de type `int`.

Dès que ce nouveau stockage est achevé, il serait intéressant de pouvoir facilement se débarrasser de l'objet `Integer`, qui a juste servi de « passerelle » entre le String et l'`int`.

Le petit code qui suit transforme l'argument String reçu lors de l'exécution du programme, par la ligne de commande indiquée ci-après, en un véritable entier correspondant. Il vous permettra également de comprendre pourquoi la méthode `main` de Java doit inclure obligatoirement un vecteur de String comme argument. Il s'agit en effet d'arguments qu'il est possible d'indiquer lors de l'exécution du programme (par exemple, le nom d'un fichier d'input…). Le passage de ces arguments se fait lors de l'instruction d'exécution du programme. Dans l'exemple, l'argument `5` est transmis comme le premier élément du vecteur de String et est ensuite transformé en l'entier « 5 ».

Ligne de commande : java ObjetInterimaire 5

```
public class ObjetInterimaire {
  public static void main(String args[]) {
    Integer unEntierInterimaire = new Integer(args[0]);
    /* On récupère le premier String passé en argument par args[0] */
    int a = unEntierInterimaire.intValue(); //transformation
    /* la méthode intValue() appliquée sur l'objet Integer permet d'en
     * récupérer la valeur entière, à ce stade-ci, l'objet
     * unEntierInterimaire n'est plus utile et pourrait être supprimé */
    System.out.println(args[0] + " s'est transforme en " + a);
  }
}
```

Résultat

```
5 s'est transforme en 5
```

Mémoire pile

Mais ce qui est vrai des objets l'est et l'a toujours été de n'importe quelle variable informatique, qui n'aurait de rôle à jouer que pendant un court laps de temps, et à l'occasion d'une fonctionnalité bien précise. Considérez le petit programme suivant, qui serait écrit de manière très semblable dans pratiquement tous les langages informatiques, dans lequel une méthode s'occupe de calculer, à partir de trois arguments reçus, les deux bases et la hauteur, la surface d'un trapèze.

Nous avons déjà abordé ce type de mécanisme dans le chapitre 6. Comme indiqué dans la figure 9-1, durant l'exécution de cette méthode, cinq variables intermédiaires vont se créer et disparaîtront aussitôt l'exécution terminée. D'abord, lors de l'appel de la méthode, trois variables nouvelles seront nécessaires pour stocker les trois dimensions du trapèze. Si ces variables existent déjà à l'extérieur de la méthode, elles seront purement et simplement dupliquées, pour être installées dans ces variables intermédiaires. Ensuite, pendant l'exécution de la méthode, une quatrième variable intermédiaire, `surface`, est créée, qui permettra de stocker le résultat jusqu'à la fin de cette exécution. Si la surface est calculable, une cinquième et dernière variable intermédiaire : `somBases`, permettra de stocker temporairement une valeur intermédiaire, dont l'usage, un peu forcé ici, permet en général une meilleure lisibilité du programme, et une algorithmique plus sûre, car décomposée en une succession d'étapes plus simples.

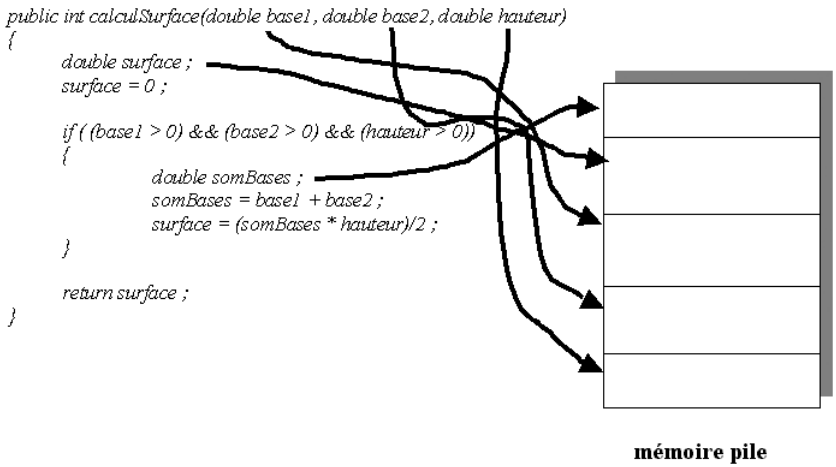


Figure 9-1

Illustration de l'existence de cinq variables temporaires utiles au calcul de la surface d'un trapèze.

Il vous paraîtra évident qu'une fois la méthode achevée, toutes ces variables doivent disparaître de la mémoire pour laisser la place à d'autres, et sans qu'on ne les y invite. C'est ce qu'elles feront dans pratiquement tous les langages, et ce le plus simplement du monde. Ces variables sont stockées, comme indiqué dans la figure, dans une mémoire dite mémoire « pile » (et qui ne s'use que si l'on s'en sert). Le principe de fonctionnement de cette mémoire est dit « LIFO » (dernier dedans premier dehors, essayez en anglais et vous comprendrez pourquoi ce fonctionnement n'a pas été dénommé « DDPD »). Dans tout code, un bloc d'instructions, encadré par les accolades, délimite également la portée des variables.

Dans une informatique séquentielle traditionnelle (nous verrons une autre solution à cela lorsque nous discuterons du « multithreading » dans le chapitre 17), un bloc d'instructions ne sera jamais interrompu. Quand un bloc se termine, les variables du dessus de la pile disparaissent tout naturellement (car elles ne sont utilisables qu'à l'intérieur de ce bloc), alors que, lorsqu'un bloc s'entame, les nouvelles variables s'installent au-dessus de la pile. Aucune recherche sophistiquée n'est nécessaire pour retrouver les variables à supprimer. Ce seront toujours les dernières à s'être installées sur la pile. De même, de cette façon, aucun gaspillage n'est possible, et aucune zone de mémoire inoccupée peut se trouver, comme un petit village gaulois, perdu au milieu de zones occupées.

Gestion par mémoire pile

Ce système de gestion de la mémoire est donc extrêmement ingénieux, car il est fondamentalement économe, gère de façon adéquate le temps de vie des variables intermédiaires par leur participation dans des fonctionnalités précises, garde rassemblées les variables qui agissent de concert, et synchronise le mécanisme d'empilement et de dépilement des variables avec l'emboîtement des méthodes.

Ce système de gestion de mémoire est très efficace pour des objets essentiellement intérimaires. Il l'est tant et si bien que C++ et C# l'ont préservé pour la gestion de la mémoire occupée par certains objets (les trois autres, quant à eux, l'ont interdit pour les objets). Idéalement, dans les deux premiers langages, vous utiliserez ce mode de gestion pour des objets dont vous connaissez à l'avance le rôle intermittent qu'ils sont appelés à jouer. Par exemple, en C++, lorsque vous créez un objet `o1` de la classe `01`, au moyen de la simple instruction :

`O1 o1`, n'importe où dans le code, sans l'utilisation du `new` et de pointeur, vous installez d'office l'objet `o1` dans la pile. Cet objet disparaîtra dès que se fermera l'accolade dont l'ouverture précède juste sa création.

De même, si vous passez un objet comme argument, automatiquement un nouvel objet sera créé, copie de celui que vous désirez passer. Une différence clé avec Java, Python et PHP 5 est qu'étant donné qu'il s'agit de la copie du référent et non pas de l'objet, la méthode, dans ces trois langages, agira bien sur l'objet original et non pas sur une copie toute fraîche, mais destinée à disparaître une fois la méthode terminée, comme en C++ et C#. Comme illustré par les codes qui suivent, il est donc possible en C# et C++ de bénéficier du même mode de gestion de mémoire pile des variables non-objets, et ce pour les objets.

En C++

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
public:
  O1() /* constructeur */ {
    cout << "un nouvel objet O1 est cree" << endl;
  }
  O1(const O1 &uneCopieO1) /* constructeur par copie */{
    cout << "un nouvel objet O1 est cree par copie" << endl;
  }
  ~O1() /* destructeur */{
    cout <<"aaahhhh ... un objet O1 se meurt ..." << endl;
  }
  void jeTravaillePourO1() {}
};
void usageO1(O1 unO1){
  unO1.jeTravaillePourO1();
}
int main(int argc, char* argv[]){
  O1 unO1; /* je crée un objet O1 */
  usageO1(unO1); /* la méthode reçoit une copie de cet objet */
  return 0;
}
```

Résultat

```
un nouvel objet O1 est créé
un nouvel objet O1 est créé par copie
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O1 se meurt...
```

Il faut, pour comprendre ce code, découvrir l'existence de deux nouvelles méthodes particulières, appelées le `constructeur par copie` et le `destructeur`. La première est appelée, automatiquement, dès qu'un objet se trouve dupliqué, notamment lors du passage d'argument. Elle permet, comme nous le comprendrons mieux dans les chapitres 10 et 14, de transformer une copie de surface en une copie profonde. Ici, ce constructeur se borne à signaler qu'on fait appel à lui.

Le destructeur, quant à lui, est une méthode appelée, automatiquement, dès la destruction d'un objet. Cette méthode ne peut recevoir d'argument car le programmeur n'est pas à l'origine de son appel. Là aussi, nous

comprendrons mieux l'importance de son rôle par la suite et dès le prochain chapitre. Elle est appelée juste avant la destruction de l'objet et permet de libérer certaines ressources référées par celui-ci avant de le faire disparaître. Ici, de même, ce destructeur se borne à signaler son appel. On voit que deux objets sont créés et détruits dans l'exécution de ce code, sans qu'il soit nécessaire de les détruire par une instruction explicite. Le second est créé lors du passage comme argument du premier. Il en est une copie. Toute cette mémoire est gérée par un système de pile, et les objets disparaissent dès la fermeture des accolades, le premier à la fin de la procédure `usageO1()`, le second à la fin du programme.

En C#

```
using System;
public struct O1 /* ATTENTION ! On utilise une structure plutôt qu'une classe */{
  private int a;

  public void jeTravaillePourO1() {
    a = 5;  // modifie l'attribut
  }
  public void donneA(){
    Console.WriteLine("la valeur de a est: " + a);
  }
}
public class TestMemoirePile {
  public static void Test(O1 unO1){
    O1 unAutreO1 = new O1();
    unAutreO1.jeTravaillePourO1();
    unO1.jeTravaillePourO1();
  } // la copie de unO1 et l'objet unAutreO1 disparaissent ici.
  public static void Main(){
    O1 unO1 = new O1();
    unO1.donneA();
    Test(unO1);
    unO1.donneA(); /* On retrouve la valeur de l'attribut a de
        l'objet de départ, malgré le passage comme argument dans la méthode Test() */
  }
}
```

Résultat

```
la valeur de a est : 0
la valeur de a est : 0
```

Dans le code C# qui précède, nous utilisons une structure en lieu et place de classe. Cela nous permet de traiter les objets issus de ces structures exactement comme n'importe quelle variable de type prédéfini. Notez qu'aucun destructeur ne peut être déclaré dans une structure d'où son absence dans notre code ici.

Ainsi, dans le code, trois objets instance de la structure `O1` sont créés. L'un des trois est créé lors du passage par argument, et on constate que la modification de son seul attribut `a` n'affecte pas l'objet original dont il n'est qu'une simple copie. Tant la copie passée par argument que les deux autres objets créés comme variable locale de la méthode `Test(O1)` et de la méthode `Main()` disparaîtront également dès la fermeture des accolades.

Structure en C#

En C#, les objets issus d'une structure sont traités directement par valeur, dans la mémoire pile, et sans référent intermédiaire. Ils le sont comme n'importe quelle variable de type prédéfini. Les structures sont utilisées en priorité pour des objets que l'on veut et que l'on sait temporaires. Un constructeur par défaut y est prévu et donc on ne peut le surcharger en en définissant explicitement un autre. Plus important encore, les structures ne peuvent hériter entre elles, bien qu'elles héritent toutes de la classe Objet. En revanche, elles peuvent implémenter des interfaces. Tout cela s'explique aisément lorsqu'on sait que les structures sont exploitées par valeur et non par référent, et que les mécanismes d'héritage et de polymorphisme sont plus faciles à réaliser pour des objets uniquement adressés par leur référents. Il est plus facile de s'échanger les référents que les objets dans leur entièreté.

C# et Anders Hejlsberg

Se retrouver au côté de Bill Gates en février 2002 à San Francisco, pour annoncer la mise sur le marché d'un nouveau logiciel Microsoft, Visual Studio .Net, présenté comme révolutionnaire car colonne vertébrale de toute une stratégie à venir et dénommée .Net, n'est pas donné au premier quidam venu. Anders Hejlsberg, de fait, n'en est pas un. Concepteur principal du langage C#, innovation capitale dans l'environnement de programmation Microsoft, Hejlsberg n'en est pas à son premier coup de maître. Danois d'origine, pays décidément fournisseur de grandes sirènes de la programmation, avant de rejoindre Microsoft en 1996, il passe quelques années chez Borland comme créateur du Turbo Pascal et en tant que leader de l'équipe à l'origine de Delphi (un autre langage OO bien connu). Chez Microsoft, il prend une part active au développement du Visual J++, habillage Microsoft de Java. On connaît les déboires que connu ce langage, hybride très habile de la syntaxe de Java avec les librairies de Microsoft, et qui irrita Sun au point de mettre la justice sur le coup. Mieux valait pour Microsoft renommer la main basse effectuée sur Java, afin de donner le jour à C# (prononcé C Sharp). Le « J » s'est, comme par magie, transformé en « C ».

Évidemment, le nom prête à plaisanteries et elles ne manquent pas. Ce langage s'est déjà vu traité de Visual J- - ou de C bémol. C'est sans doute sa ressemblance avec Java qui le rend le plus vulnérable à ces agressions. Et nous savons la profonde et indéfectible amitié qui lie Bill Gates à Scott McNeal, dirigeant de Sun. Dans la bouche de ce dernier .Net devient .Not, .Not yet ou .Nut. Rien d'étonnant à qui déclare aussi que « dans cette guerre sans merci, nous récupèrerons chaque développeur et ne le laisserons pas s'abandonner du côté sombre ». Plusieurs fois dans notre ouvrage, nous constatons, parfois avec agacement, ces similitudes dont se défend pourtant notre auteur (stratégie et marketing obligent). En effet, dans les premiers écrits consacrés à ce nouveau langage, il était très difficile de trouver une simple mention à Java. Le langage était perçu comme une évolution naturelle de C++, mais l'absence d'allusion à Java ramenait celui-ci à un statut de véritable « chaînon manquant ».

S'il est vrai que C# s'est moins éloigné de C++ que Java ne l'a fait (on y retrouve davantage de types de données communs à C++, des objets stockables en mémoire pile, une prédominance du typage statique, et on peut même y intégrer, à ses risques et périls, des pointeurs C++), il n'en reste pas moins vrai que son plus proche voisin demeure Java et non pas C++. Et pour cause, le langage récupère la cohérence OO héritée de Smalltalk, mais, tout comme Java, base cet habillage OO sur une syntaxe C. On y retrouve le ramasse-miettes et une interprétation du code plutôt qu'une compilation, interprétation qui se transforme, cependant, en une véritable compilation dès la première exécution du code (les performances sont alors améliorées). Ce niveau intermédiaire permet une communication facilitée entre différents langages de programmation.

Est-il meilleur que Java ? Voilà le type même de question à laquelle il est impossible de répondre, et cela l'est également pour tous les langages que nous traitons dans ce livre. L'Italie est-elle meilleure que la France ? La paëlla est-elle meilleure que le couscous ? C# a pour lui d'apparaître cinq ans après Java et de pouvoir puiser çà et là ce qui se fait de mieux dans les étals de plusieurs langages. Sans doute a-t-il passé un peu plus de temps au rayon Sun et, postérieur à Java, il a pu éviter certaines maladresses de celui-ci dont se plaignent les programmeurs et que Java ne peut supprimer par souci de compatibilité avec l'existant. C# intègre, par exemple, des aspects de VB, comme les mécanismes d'accès aux attributs. Son jeune âge lui permet aussi d'incorporer des éléments technologiques plus modernes. Il en va ainsi de la totale prise en compte d'XML, langage universel de description de contenu de documents publiés sur le Web. Il est possible, à partir du code, de générer très facilement une description de son contenu en XML. Hejlsberg le décrit comme le premier langage facilitant véritablement la programmation à base de composants, bien qu'il reste généralement très évasif sur ce qu'il entend par là, et en quoi ni Java ni d'autres ne pourraient servir à la programmation de ces mêmes composants.

Il est clair que, quitte à s'embarquer sur la nouvelle plate-forme Microsoft de développement Internet, le langage C# apparaît comme un outil de développement de choix (il est d'ailleurs recommandé par Microsoft). Nous verrons dans le chapitre 16 sa prise en compte très simple et très naturelle des services web, version XML des objets distribués communiquant à travers le Web. De fait, Hejlsberg présente toujours son langage comme partie intégrante de .Net, en le plébiscitant comme un des éléments clés de cette énorme boîte à outils de développement Internet. .Net permet, en effet, un développement facilité et transparent d'applications distribuées, par l'entremise des services Web générés automatiquement à partir des codes sources. .Net, dont la vocation première est de faciliter la conception d'applications distriuées sur le Web par l'entremise d'ASP.Net, lorsque ces applications se parlent via un browser, ou par service web quand les objets communiquent directement par envoi de message d'une application à l'autre, a enrichi la description sémantique de ces interactions par l'utilisation abondante et largement automatisée du langage XML. Cette intégration a permis à Microsoft de prendre quelques longueurs d'avance par rapport à son concurrent direct, Sun.

Malgré son « interprétabilité » (commune à Java), C# pour l'instant ne tourne que sur Windows. Microsoft parle d'universalité de cette plate-forme mais dans un sens nouveau. La version interprétable du langage (CLR – Common Langage Runtime) est partageable avec de nombreux autres langages supportés par .Net (vingt-deux à ce jour), comme C++, VB .Net, Jscript, Cobol, Eiffel (d'où la participation de Meyer), Perl, Python, Smalltalk et d'autres à venir. Cela permet à un code C# d'hériter éventuellement d'une classe préalablement développée en VB .Net, et d'envoyer un message à une classe développée en Eiffel. Cela est possible, car tous ces langages se conforment à une CLS (Common Langage Specification – d'où, de fait, la nouvelle version de VB) qui permet de passer de l'un à l'autre. Là où Java est mono-langue mais multi- plates-formes, .Net est multi-langues mais mono-plate-forme. Une tentative actuelle de standardisation de C# est en cours. Parlons aussi du projet plus récent Mono destiné à pourvoir une version Open Source de .Net qui sera susceptible de tourner sur des plates-formes aussi variées que Linux, Solaris, Mac OS X.

La version 2 de .NET, langage et environnement, s'était illustrée par la prise en charge de la généricité traitée au chapitre 21. La version 3 de .NET connaît une autre avancée majeure, la bibliothèque Linq (décrite au chapitre 19) qui propose une nouvelle méthode – une de plus – pour améliorer la correspondance entre le monde des bases de données relationnelles et l'orienté objet. Linq propose un langage d'interrogation, inspiré du SQL, capable de s'interfacer indifféremment avec des collections d'objets, ces mêmes objets étant stockés soit sous forme XML soit dans une base de donnée relationelle.

Visual C++.Net

C++ étant largement étudié dans cet ouvrage, nous nous en voudrions de ne pas dire un petit mot sur Visual C++.Net, la nouvelle mouture de ce langage, une version assez radicalement remaniée grâce à son intégration à .Net et à cause de la nécessité de se rendre compatible aux vingt-et-un autres langages supportés par la plate-forme. Ce nouveau langage est étonnament puissant car, par exemple, il combine les systèmes de gestion mémoire par ramasse-miettes (on parle alors de `_gc_class` plutôt que de `class`, et toute la sophistication consiste à mêler ces deux types de class et la gestion mémoire différente qui s'y rapporte) et par instruction directe du programmeur. De même que C++ ne présente pas la même offre en librairies que Java (Multithread, GUI, bases de données), ce nouvel arrivant intègre idéalement toutes les librairies .Net, ce qui le rend aussi riche en services et librairies que Java. (Ces librairies sont de fait communes à tous les langages de la plate-forme. Vous pouvez en voir quelques-unes – Ado.Net, les services web ou les GUI – à l'œuvre dans certains chapitres de ce livre.)

Il est difficile, vu son jeune âge, de vous donner une liste définitive de manuels de programmation C#. Comme introduction très rapide et très économique au langage, on peut citer :

- *Le Langage C#*, Christine EBEHARDT, Campus Press
- *Introduction à C#*, Pierre-Yves SAUMONT et Antoine MIRECOURT, Osman Eyrolles MultiMedia.
- Une description rapide mais approfondie (parfaite pour les programmeurs Java) se trouve dans : *C# Essentials*, Ben ALBARHI, Peter DRAYTON et Brad MERRIL, O'Reilly.
- DEITTEL et DEITELL ne sont évidemment pas en reste et ont récemment publié *C# how to program* dans la collection Prentice Hall ainsi qu'une version plus avancée ; C# for Experienced Programmers.
- Le très bon *Programming C#* de Jesse LIBERTY chez O'Reilly et *C# And Object Orientation* de John HUNT chez Springer.

Et enfin pour C# dans le contexte .Net :
- *Microsoft .Net for Programmers*, Fergal GRIMES, Manning
- *Beginning Asp.Net using C#*, Chris ULLMAN, Chris GOODE, Juan T. LLIBRE et Ollie CORNES, Wrox.
- *Microsoft. NET for Programmers*, Fergal GRIMES, Manning.

Disparaître de la mémoire comme de la vie réelle

Ce mode de gestion de la mémoire pile est intimement lié à une organisation procédurale de la programmation, où le programme est décomposé en procédures ou en blocs imbriqués, lesquels nécessiteront, uniquement pendant leur déroulement, un ensemble de variables, qui seront éliminées à la fin. Nous avons vu que la programmation OO se détache de cette vision, en privilégiant les objets aux fonctions. Il est, en conséquence, tout aussi important de détacher le temps de vie des objets de leur participation à certaines fonctions précises. L'esprit de l'OO est qu'un objet devient encombrant si, dans le scénario même que reproduit le programme, l'objet réel, que son modèle informatique « interprète », disparaît tout autant de la réalité. Dans le petit écosystème vu précédemment, la proie disparaît quand elle se fait manger par le prédateur, l'eau disparaît quand sa quantité devient nulle.

Représentez-vous tous ces jeux informatiques, dans lesquels des balles apparaissent et disparaissent, des avions explosent, des héros meurent, des footballeurs quittent le terrain. À chaque fois, l'objet représenté disparaît, tant et si bien que son élimination de la mémoire est même souhaitée, pour permettre à un nouvel objet d'exister et prendre sa place. Il est bien plus difficile d'organiser cette gestion de la mémoire par un mécanisme de pile car, une fois l'objet créé, son temps de vie peut transcender plusieurs blocs fonctionnels, pour ne finalement disparaître, éventuellement de manière conditionnelle, dans l'un d'entre eux (et pas du tout automatiquement dans le bloc où il fut créé). L'OO permet aux objets de vivre bien plus longtemps (et ils vous en remercient) et, surtout, rend leur élimination indépendante des fonctions qui les manipulent, mais plus dépendante du scénario qui se déroule, aussi inattendu soit-il.

La vie des objets indépendante de ce qu'ils font

L'orienté objet, se détachant d'une vision procédurale de la programmation, tend à rendre indépendante la gestion mémoire occupée par les objets de leur participation dans l'une ou l'autre opération. Cette nouvelle gestion mémoire résultera d'un suivi des différentes transformations subies par l'objet et sera, soit laissée à la responsabilité du programmeur, soit automatisée.

Mémoire tas

Tous les langages OO permettent donc un mode de création et de destruction d'objets autrement plus flexible que la mémoire pile. En C++ et C#, ce nouveau mode est en complément de la mémoire pile. En Java, PHP 5 et Python, ce nouveau mode est le seul possible pour les objets, la mémoire pile restant, en revanche, la seule possibilité pour toutes les autres variables de type prédéfini ou primitif. Dans ce mode plus flexible et en C++, C, PHP 5 et Java, tous marqués à vie par leur précurseur, le C, on peut créer les objets n'importe où dans le programme par le truchement du `new` (en Python, `new` n'est plus nécessaire). Ils seront créés n'importe quand et pourront être installés partout où cela est possible dans la mémoire, d'où la nécessité d'un référent qui connaisse leur adresse et permette de les retrouver et les utiliser. Mais comment fera-t-on disparaître un objet ? Simplement, quand la « petite histoire » que raconte le programme l'exige ? De nouveau, il est nécessaire de différencier deux politiques : celle très libérale du C++, qui laisse au seul programmeur le soin de décider de la vie et de la mort des objets, et le mode étatisé des quatre autres langages, qui s'en occupe pour vous, en arrière-plan.

C++ : le programmeur est le seul maître à bord

En C++, vous pouvez, n'importe où dans un programme, supprimer un objet qui a été créé par `new`, en appliquant sur son référent l'instruction `delete`. Vous devenez les seuls maîtres à bord, et, à ce titre, capable du meilleur comme du pire. Pour le meilleur, on vous fait confiance, malheureusement, pour le pire, on vous conserve cette confiance. Ainsi, voici deux scénarios catastrophes, toute proportion gardée bien entendu, que les programmeurs C++ reconnaîtront aisément, même s'ils s'en défendent.

Un premier petit scénario catastrophe en C++

```
#include "stdafx.h"
#include "iostream.h"
class O1{
private:
  int a;
public:
  O1() /* constructeur */{
    a = 5;
    cout << "un nouvel objet O1 est cree" << endl;
  }
  O1(const O1 &uneCopieO1) /* constructeur par copie */{
      cout << "un nouvel objet O1 est cree par copie" << endl;
  }
  ~O1() /* destructeur */{
    cout <<"aaahhhh ... un objet O1 se meurt ..." << endl;
  }
  void jeTravaillePourO1() {
    cout << "a vaut: "<< a << endl;
  }
};
void jeTueObjet(O1 *unO1){
  delete unO1; // on efface l'objet O1
}
void jeCreeObjet(){
  O1 *unO1 = new O1();
  jeTueObjet(unO1);
  unO1->jeTravaillePourO1(); //l'objet a disparu bien que son utilisation reste parfaitement
  ➥possible.
}
int main(int argc, char* argv[]){
  jeCreeObjet();
  return 0;
}
```

Résultats

```
un nouvel objet O1 est créé
aaahhhh... un objet O1 se meurt...
a vaut : - 572662307
```

Un même objet `unO1` est référencé deux fois. Dans la procédure `jeCreeObjet()` (on parlera de procédure ou de fonction car elle est définie en dehors de toute classe), on crée d'abord l'objet `unO1`, et puis on le passe en argument de la méthode `jeTueObjet()`, qui s'empresse de l'effacer. Mais alors qu'il est éliminé par la méthode `jeTueObjet()`, il est encore référencé dans la méthode `jeCreeObjet()` par le référent `unO1`. Comme l'objet référé par ce référent a disparu, ce dernier se mettra à référer n'importe quoi dans la mémoire, avec toutes les mauvaises surprises dont les programmeurs du C++ sont friands.

Vous voyez, par exemple, qu'au lieu d'afficher la valeur `5`, ce à quoi on aimerait s'attendre, c'est une valeur complètement imprévue qui apparaît. Rien dans la compilation du programme n'a pu prévenir ce dysfonctionnement. Évidemment, dans ce petit code, l'endroit où est créé l'objet est tellement proche de l'endroit où celui-ci est détruit qu'une telle situation vous semble parfaitement improbable. Détrompez-vous ! Dans un code plus grand et bien plus réparti entre les programmeurs, un de ces derniers, dans l'écriture de sa méthode, aura tout loisir de détruire un objet encore utile à un tas d'autres programmeurs. Nous avons vu que l'adressage indirect, permettant à un même objet d'être référé de multiples fois (dans des contextes différents), est un mécanisme inhérent au paradigme objet. Chaque référent devient alors disponible pour effacer l'objet, quand bien même les autres référents, tout perdus qu'ils soient, persisteraient à y faire référence ! Les référents fous ou pointeurs fous sont légion en C++, et aucune voiture d'ambulanciers ne se charge de les récupérer dans la mémoire. Un autre scénario tout aussi dramatique est celui qui consiste à effacer plusieurs fois un même objet par la répétition de l'instruction `delete` sur un même référent. Les référents fous ou pointeurs fous sont légion en C++, et aucune voiture d'ambulanciers ne se charge de les récupérer dans la mémoire. Un autre scénario tout aussi dramatique est celui qui consiste à effacer plusieurs fois un même objet par la répétition de l'instruction `delete` sur un même référent.

La mémoire a des fuites

Rappelez-vous le petit laïus moralisateur au début du chapitre, vous incitant à ne pas gaspiller la mémoire des ordinateurs, sauf à la jeter dans un sac poubelle prévu à cet effet. Il est très fréquent, dans les langages où vous êtes responsables de la gestion mémoire, de laisser traîner des objets devenus inaccessibles, donc parfaitement encombrants. On parle alors de « fuite de mémoire ». Ce scénario porte moins à conséquence que le précédent. C'est même le scénario parfaitement inverse car, maintenant, alors que les objets existent encore, ils sont devenus hors de portée. Ils ralentissent le code et font chuter les performances, mais n'occasionnent rien de totalement imprévisible.

Second petit scénario catastrophe en C++

```
#include "stdafx.h"
#include "iostream.h"
class O2{
public:
   O2(){
     cout << "un nouvel objet O2 est cree" << endl;
   }
  ~O2() /* destructeur */{
    cout <<"aaahhhh ... un objet O2 se meurt ..." << endl;
   }
};
class O1{
private:
  O2 *monO2; /* on agrège un objet O2 dans O1 */
public:
   O1() /* constructeur */{
```

```
      cout << "un nouvel objet O1 est cree" << endl;
      monO2 = new O2(); /* on crée ici l'objet O2 */
    }
    ~O1() /* destructeur */{
     cout <<"aaahhhh ... un objet O1 se meurt ..." << endl;
    }
};
int main(int argc, char* argv[]){
  O1 *unO1 = new O1();
  unO1 = new O1(); /* on ré-utilise le même référent */
  delete unO1;
  return 0;
}
```

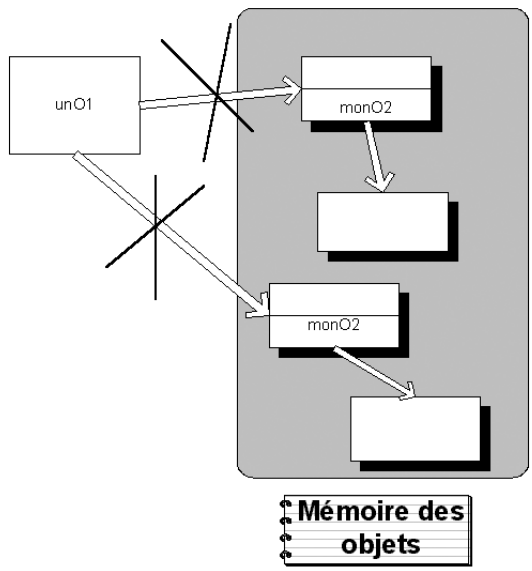
Résultats

```
un nouvel objet O1 est créé
un nouvel objet O2 est créé
un nouvel objet O1 est créé
un nouvel objet O2 est créé
aaahhhh... un objet O1 se meurt...
```

La figure 9-2 montre ce qui se passe dans la mémoire des objets lorsque le programme procède à la destruction du dernier objet `O1` référé. Trois objets continueront à encombrer la mémoire inutilement, mémoire gaspillée et irrécupérable. La première raison en est l'utilisation du même référent `unO1` pour les deux objets créés. Alors que deux objets `O1` sont créés, seul le second sera accessible, car le même référent que celui utilisé pour le premier objet est exploité à nouveau. De ce fait, comme l'attribut `monO2 du` premier objet `O1` pointe vers un second objet, les deux objets occuperont inutilement la mémoire. Lorsque grâce au `delete`, vous effacez le second `O1`, en fait vous n'effacez que cet objet et le référent vers son objet `O2`. Mais si vous omettez d'effacer

Figure 9-2

Le déroulement en mémoire des codes C++ et Java commentés dans le texte. Le référent « unO1 » pointe d'abord sur un premier objet O1 (chaque objet O1 pointe à son tour sur un objet O2) pour ensuite se mettre à pointer sur un autre objet O1 avant de passer à null. En C++, seul le troisième objet dans la mémoire sera effacé. En Java, C# et Python grâce au comptage des référents et au ramasse-miettes, tous les objets finiront par être effacés de la mémoire.



l'objet `O2` également (ce que vous pouvez faire par un mécanisme qui sera détaillé dans le prochain chapitre, et qui consiste à redéfinir le destructeur), celui-ci, à son tour, occupera inutilement la mémoire.

En substance, un langage comme C++, qui vous espère adulte et baptisé en matière de gestion de mémoire, a tendance à quelque peu surestimer ses programmeurs. Et ceux-ci se retrouvent, soit avec des référents fous, qui se mettent à référer de manière imprévisible tout et n'importe quoi dans la mémoire, soit avec des objets perdus, comme des satellites égarés dans l'espace, sans aucun espoir de récupération.

En Java, C#, Python et PHP 5 : la chasse au gaspi

D'autres langages, comme Java, C#, Python et PHP 5, se montrent moins confiants quant à vos talents de programmeurs, et préfèrent prévenir que guérir. Ils partent de l'idée toute simple qu'un objet n'est plus utile dès lors qu'il ne peut plus être référencé. Un objet devenu inaccessible ne demande qu'à vous restituer la place qu'il occupait. L'idéal serait donc de réussir à vous débarrasser, à votre insu (mais tout à votre bénéfice), pendant l'exécution du programme, des objets devenus encombrants. Une manière simple est de rajouter, comme attribut caché de chaque objet, un compteur de référents, et de supprimer tout objet dès que ce compteur devient nul. En effet, un objet débarrassé de tout référent est inaccessible, donc inutilisable, et donc bon à jeter.

Si vous vous repenchez sur les deux petits codes présentés précédemment, vous verrez que ce seul mécanisme vous aurait évité, d'abord, le référent fou (puisque vous ne pouvez vous-même effacer un objet, un référent fou devient impossible), ensuite la perte de mémoire. En effet, le premier objet `O1` disparaît car il n'est plus référencé par le référent `unO1`. Avec lui, disparaît également le référent vers l'objet `O2`, entraînant donc ce dernier dans sa perte. Par exemple, le petit code Java suivant, équivalent, dans l'esprit, au code C++ précédent, vous montre que tous les objets inaccessibles seront en effet détruits. La méthode `finalize()` joue, en substance, le même rôle que le destructeur en C++, et s'exécute lors de la destruction de l'objet. Nous reviendrons sur son rôle dans les prochains chapitres.

En Java

```
class O2{
  public O2() /* constructeur */{
    System.out.println("un nouvel objet O2 est cree");
  }
  protected void finalize () /* le destructeur */{
    System.out.println("aaahhhh ... un objet O2 se meurt ...");
  }
}
class O1{
  private O2 monO2; /* on agrège un objet O2 dans O1 */

  public O1() { /* constructeur */
    System.out.println("un nouvel objet O1 est cree");
    monO2 = new O2(); /* on crée ici l'objet O2 */
  }
  protected void finalize() { /* destructeur */
    System.out.println("aaahhhh ... un objet O1 se meurt ...");
  }
}
```

```
public class TestFuiteMemoire{
  public static void main(String[] args){
    O1 unO1 = new O1();
    unO1 = new O1(); /* on ré-utilise le même référent */
    unO1 = null;
    System.gc(); /* appel explicite du garbage-collector */
  }
}
```

Résultats

```
un nouvel objet O1 est créé
un nouvel objet O2 est créé
un nouvel objet O1 est créé
un nouvel objet O2 est créé
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
```

En C#

```
using System;
class O2{
  public O2() /* constructeur */{
    Console.WriteLine("un nouvel objet O2 est cree");
  }
    ~ O2() /* le destructeur */{
    Console.WriteLine("aaahhhh ... un objet O2 se meurt ...");
  }
}
class O1{
  private O2 monO2; /* on agrège un objet O2 dans O1 */
  public O1() { /* constructeur */
    Console.WriteLine("un nouvel objet O1 est cree");
    monO2 = new O2(); /* on crée ici l'objet O2 */
  }
    ~ O1() { /* destructeur */
    Console.WriteLine("aaahhhh ... un objet O1 se meurt ...");
  }
}
public class TestFuiteMemoire{
  public static void Main(){
    O1 unO1 = new O1();
    unO1 = new O1(); /* on réutilise le même référent */
    unO1 = null;
    GC.Collect(); /* appel explicite du garbage-collector */
  }
}
```

Le code C#, parfaitement équivalent au code Java, est présenté de manière à indiquer les quelques différences d'écriture avec Java, notamment dans la syntaxe du destructeur, qui rappelle plutôt celle du C++.

En PHP 5

Rien de bien original dans la version PHP 5 du même code qui produira, là encore, le même résultat.

```
<html>
<head>
<title> Gestion mémoire des objets </title>
</head>
<body>
<h1> Gestion mémoire des objets </h1>
<br>

<?php
   class O2 {

           public function __construct() {
                       print ("un nouvel objet O2 est cree <br> \n");
           }

           public function __destruct () { // déclaration du destructeur
                       print ("aaahhhh .... un objet O2 se meurt ... <br> \n");
           }

   }

   class O1 {
           private $monO2;

           public function __construct() {
                       print ("un nouvel objet O1 est cree <br>\n");
                       $this->monO2 = new O2();


   }
           public function __destruct () {
                       print ("aaahhhh .... un objet O1 se meurt ... <br> \n");
           }
   }

   $unO1 = new O1();
   $unO1 = new O1();
   $unO1 = NULL;

   ?>

</body>
</html>
```

Le ramasse-miettes (ou garbage collector)

On peut voir qu'au contraire du C++, dans les trois autres langages, tous les objets encombrants sont effacés :

- le premier objet `O1`, car on réutilise son référent pour la création d'un autre objet ;
- le second car on a mis son référent à `null`.

Cette dernière instruction est utile lorsque l'on cherche à se débarrasser d'un objet devenu encombrant : il suffit d'assigner la valeur `null` à son référent. À la différence de C++, les autres langages n'autorisent pas une suppression d'objet par une simple instruction. La dernière instruction du programme ne se rencontre en général pas, car il y est explicitement fait appel à l'effaceur d'objets : le garbage collector. En général, cet appel se fait à une fréquence soutenue et calibrée par défaut par la machine virtuelle de Java, C#, Python ou PHP 5, dès que la mémoire RAM commence à être sérieusement occupée. Ce calibrage suffit dans la plupart des cas, mais vous avez néanmoins la possibilité d'interférer directement avec ce mécanisme, comme indiqué dans les codes.

Le mécanisme responsable de la découverte et de l'élimination des objets perdus s'appelle, en effet, le garbage collector, traduisible par « camion-poubelle » ou « ramasse-miettes ». Le ramasse-miettes passe en revue tous les objets de la mémoire avec pour mission d'effacer ceux qui possèdent un compteur de référents nul. En général, il s'exécute en parallèle (c'est-à-dire sur un thread à part) avec votre programme. (Nous découvrirons le multithreading dans le chapitre 17.)

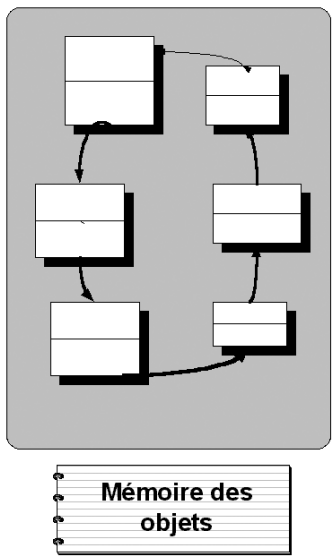
Souvent, celui-ci se déclenchera naturellement, lorsqu'on commence à remplir la mémoire de manière conséquente. Il est quelquefois paramétrable, selon que vous le souhaitiez hyperactif et donc très concentré sur les économies à réaliser dans la mémoire, ou plus laxiste. Il est clair qu'un compromis subtil est à rechercher ici, car souvent les économies mémoire permettent une accélération du programme. Mais si le prix à payer pour récupérer cette précieuse mémoire est un ralentissement encore plus important du programme, occasionné par le fonctionnement simultané du ramasse-miettes et de votre programme, vous en percevez aisément le ridicule (qui ne tue ni les êtres humains ni les objets malheureusement).

Des objets qui se mordent la queue

Un seul problème subsiste et, hélas, non des moindres : ce compteur de référents peut être non nul pour certains objets, bien que ces objets en question soient non accessibles. Il s'agit de tous les objets impliqués dans des structures relationnelles présentant des cycles, comme dans la figure 9-3.

Figure 9-3

Tous les objets sont référés au moins une fois, mais le cycle entier d'objets est inaccessible.



Cette situation est plus que fréquente en OO, car il suffit par exemple que deux objets, comme notre proie et prédateur, se réfèrent mutuellement pour que cela se produise. La détection de ces cycles nécessite une très laborieuse exploration de la mémoire, qui a finalement l'effet pervers, si elle se déroule simultanément à l'exécution du programme, de ralentir celui-ci. Les langages qui ont opté pour la manière automatique de récupération de mémoire ont donc inventé des systèmes ingénieux, dont la description dépasserait le cadre de cet ouvrage, pour parer au mieux à ce problème.

Ils sont sûrs de ce qu'ils font, en ceci qu'aucun objet utile ne peut disparaître, et qu'aucun référent ne puisse être atteint de soudaine folie. Cependant, ils acceptent de ne pas être parfaits et exhaustifs, en abandonnant dans la mémoire quelques objets qui sont devenus inutiles. Par exemple, ils choisissent de ne pas systématiquement passer toute la mémoire en revue, à la recherche des objets perdus, mais uniquement de se concentrer sur les objets les plus récemment créés. Les objets dont la création récente résulte d'une utilisation temporaire pour une fonctionnalité précise seront d'excellents candidats à une élimination rapide.

Une dernière opération à réaliser, une fois la mémoire récupérée, est de ré-organiser celle-ci de façon à très facilement repérer les zones occupées et inoccupées. Cette opération aura pour effet de déplacer les objets du programme afin de les compacter dans la mémoire. Ce recompactage des objets permet d'exploiter au mieux les systèmes de mémoire hiérarchique, tels que la mémoire cache. En effet, la chance que les objets agissant de concert se trouvent localisés dans une zone voisine s'accroît, en les installant les uns à côté des autres. Il faudra, à votre insu (mais c'est autant de gagné bien sûr), changer les adresses contenues dans les référents. Ce ramasse-miettes existant en Java, C#, Python, PHP 5 et originellement dans LISP, est donc un procédé d'une grande sophistication, tentant au mieux de vous éviter les terribles méprises ou gaspillages inhérents au C++, tout en « prenant conscience » de son coût en temps calcul, et des moyens de diminuer celui-ci. La claque, quoi !

Nous finissons le chapitre en présentant la même version des codes Java, C# et PHP 5 mais cette fois-ci en Python, afin d'expliquer davantage le rôle du mot-clé `self`. Les résultats des deux versions du code sont indiqués en dessous de celui-ci.

En Python

```
import gc # imbrication dans le code des fonctionnalités du ramasse-miettes
class O2:
    def __init__(self):
        print "un nouvel objet O2 est cree"
    def __del__(self):
        print "aaahhhh ... un objet O2 se meurt ..."
class O1:
    __monO2 = None
    def __init__(self):
        print "un nouvel objet O1 est cree"
        # self.__monO2 = O2() # première version du code
        # __monO2 = O2() # deuxième version du code
    def __del__(self):
        print "aaahhhh ... un objet O1 se meurt ..."
unO1 = O1()
unO1 = O1()
unO1 = None
gc.collect()
```

Résultat de la première version du code, avec le self, comme dans les exemples Java et C#

```
un nouvel objet O1 est cree
un nouvel objet O2 est cree
un nouvel objet O1 est cree
un nouvel objet O2 est cree
aaahhhh ... un objet O1 se meurt ...
aaahhhh ... un objet O2 se meurt ...
aaahhhh ... un objet O1 se meurt ...
aaahhhh ... un objet O2 se meurt ...
```

Résultat de la deuxième version du code, sans le self

```
un nouvel objet O1 est cree
un nouvel objet O2 est cree
aaahhhh ... un objet O2 se meurt ...
un nouvel objet O1 est cree
un nouvel objet O2 est cree
aaahhhh ... un objet O2 se meurt ...
aaahhhh ... un objet O1 se meurt ...
aaahhhh ... un objet O1 se meurt ...
```

La première version du code est identique à celle des codes Java et C#. Cependant, dès que l'on supprime le `self`, le référent `__monO2` ne réfère plus l'attribut de la classe, mais un nouvel objet, variable locale du constructeur et qui se borne à disparaître dès que le constructeur a fini de s'exécuter. C'est la présence du `self` (en tout point semblable à la présence du `$this->` en PHP 5) qui permet de différencier l'appel explicite aux attributs de l'objet de la création et l'utilisation de variables locales aux méthodes. Le paramètre self est donc indispensable, comme dans la plupart des codes Python qui précèdent, dès que l'on utilise les attributs propres à l'objet. L'omettre entraîne la création et la manipulation de variables locales aux méthodes.

Afin de clarifier davantage encore la façon subtile dont Python différencie, dans ses classes variables locales, attributs de classe et attributs d'objet, le petit code suivant devrait vous être très utile.

En Python

```
class O1:
    a=0
    b=0
    c=0
    def test(self):
        a=1 #cela reste une variable locale car elle n'est liée à rien lors de son affectation
        O1.b=2 #cela reste un attribut de classe car il est référé comme tel
        self.c=3 #cela devient, grâce à la présence de self, un attribut d'objet
        print a,O1.b,O1.c
unO1 = O1()
unO1.test()
print O1.a,unO1.a
print O1.b,unO1.b
print O1.c,unO1.c
```

Résultats

```
1 2 0
0 0
2 2
0 3 #ici, on fait bien la différence entre « c » attribut de classe et « c » attribut d'objet
```

Le garbage collector ou ramasse-miettes

Il s'agit de ce mécanisme puissant, existant dans Java, C#, Python et PHP 5, qui permet de libérer le programmeur du souci de la suppression explicite des objets encombrants. Il s'effectue au prix d'une exploration continue de la mémoire, simultanée à l'exécution du programme, à la recherche des compteurs de référents nuls (un compteur de référents existe pour chaque objet) et des structures relationnelles cycliques. Une manière classique de l'accélérer est de limiter son exploration aux objets les plus récemment créés. Ce ramasse-miettes est manipulable de l'intérieur du programme et peut être, de fait, appelé ou simplement désactivé (dans les trois langages, ce sont des méthodes envoyées sur « gc » qui s'en occupent). Les partisans du C++ mettent en avant le coût énorme en temps de calcul et en performance occasionné par le fonctionnement du « ramasse-miettes ». Mais à nouveau, lorsqu'il est question de comparer le C++ aux autres langages (notez que des librairies existent qui permettent de rajouter un mécanime de garbage collector au C++), la chose s'avère délicate car les forces et les faiblesses ne portent en rien sur les mêmes aspects. C++ est un langage puissant et rapide, mais uniquement pour ceux qui ont choisi d'en maîtriser toute la puissance et la vitesse. Mettez une Ferrari dans les mains d'un conducteur qui n'a d'autres besoins et petits plaisirs que des sièges confortables, une voiture large et silencieuse ainsi qu'une complète sécurité, vous n'en ferez pas un homme heureux. Mettez un appareil photo aussi sophistiqué qu'un Hasselblad dans les mains d'un amateur qui n'a d'autres priorités que de faire rapidement et sur-le-champ des photos assez spontanées de ses vacances et les photos seront toutes ratées.

Exercices

Exercice 9.1

Expliquez pourquoi la mémoire RAM est une ressource précieuse, et pourquoi il est nécessaire de tenter au mieux de rassembler les objets dans cette mémoire.

Exercice 9.2

Dans le petit code C++ suivant, combien d'objets résideront-ils de façon inaccessible en mémoire, jusqu'à la fin du programme ?

```
#include "stdafx.h"
class O1 {
};
class O2 {
private:
  O1 *unO1;
public:
  O2()
  {
    unO1 = new O1();
  }
};
```

```
class O3 {
private:
  O2 *unO2;
public:
  O3(){
    unO2 = new O2();
  }
};
int main(int argc, char* argv[]) {
  O3 *unO3 = new O3();
  delete unO3;
  return 0;
}
```

Exercice 9.3

Dans un code équivalent en Java, tel que le code écrit ci-après, combien d'objets résideront encore en mémoire après l'appel au ramasse-miettes ?

```
class O1 {
}
class O2 {
  private O1 unO1;
  public O2(){
    unO1 = new O1();
  }
}
class O3 {
  private  O2 unO2;
  public O3(){
    unO2 = new O2();
  }
}
public class Principale {
  public static void main(String[] args) {
    O3 unO3 = new O3();
    unO3 = null;
    System.gc();
  }
}
```

Exercice 9.4

Indiquez ce que produirait le petit code C++ suivant, et expliquez pourquoi ce problème ne pourrait survenir en Java et en C# :

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
private:
  int a;
```

```
public:
  O1() {
    a = 10;
  }
  void donneA() {
    cout << a << endl;
  }
};
class O2 {
public:
  O2(){
  }
  void jeTravaillePourO2(O1 *unO1) {
    delete unO1;
  }
  void jeTravailleAussiPourO2(O1 *unO1) {
    unO1->donneA();
  }
};
int main() {
  O1* unO1 = new O1();
  O2* unO2 = new O2();
  unO2->jeTravaillePourO2(unO1);
  unO2->jeTravailleAussiPourO2(unO1);
  return 0;
}
```

Exercice 9.5

Expliquez pourquoi le code Java suivant présente des difficultés pour le ramasse-miettes :

```
class O1 {
  private O3 unO3;
  public O1(){
    unO3 = new O3(this);
  }
}
class O2 {
  private O1 unO1;
  public O2(O1 unO1) {
    this.unO1 = unO1;
  }
}
class O3 {
  private  O2 unO2;

  public O3(O1 unO1) {
    unO2 = new O2(unO1);
  }
}
```

```
class O4 {
  private O1 unO1;
  public O4() {
    unO1 = new O1();
  }
}
public class Principale2 {
  public static void main(String[] args) {
    O4 unO4 = new O4();
    unO4 = null;
    System.gc();
  }
}
```

Exercice 9.6

Quel nombre affichera ce programme C++ à l'issue de son exécution ?

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
private:
  static int morts;
public:
  static int getMorts(){
    return morts;
  }
public:
  ~O1(){
    morts++;
  }
};
int O1::morts = 0;

void essai(O1 unO1) {
  O1 unAutreO1;
}
int main(int argc, char* argv[]) {
  O1 unO1;
  for (int i=0; i<5; i++) {
    essai(unO1);
  }
  cout << O1::getMorts() << endl;
  return 0;
}
```

Exercice 9.7

Expliquez le fonctionnement du ramasse-miettes, les difficultés qu'il rencontre et les manières de l'accélérer.

Exercice 9.8

Expliquez comment et pourquoi dans la gestion mémoire des objets, C# tâche d'être un compromis parfait entre Java et C++.